# Enhancing Language Models with Structured Reasoning

Aman Madaan

CMU-LTI-24-002

March 15

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**

| | |
|---|---|
| Yiming Yang (Chair) | Carnegie Mellon University |
| Graham Neubig | Carnegie Mellon University |
| Daniel Fried | Carnegie Mellon University |
| Niket Tandon | Allen Institute for Artificial Intelligence |

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy.*

# Abstract

The rapid growth in the areas of language generation and reasoning has been significantly facilitated by the availability of user-friendly libraries wrapped around large language models. These solutions often rely on the Seq2Seq paradigm, treating all problems as text-to-text transformations. While convenient, this approach faces limitations in practical deployments: brittleness when handling complex problems, the absence of feedback mechanisms, and an inherent black-box nature hindering model interpretability.

This thesis presents techniques to address these limitations by integrating structured elements into the design and operation of language models. Structure, in this context, is defined as the organization and representation of data in systematic, hierarchical, or relational ways, along with incorporating structural constraints into the learning and reasoning processes. These elements are integrated at different model development and deployment stages: training, inference, and post-inference. During training, we present techniques for training a graph-assisted question-answering model, and discovering orders that help in effectively generating sets as sequences. In the inference stage, we present techniques for incorporating structure by leveraging code as an intermediate representation. For the post-inference stage, we introduce methods that integrate a memory to allow the model to leverage feedback without additional training.

Together, these techniques demonstrate that conventional text-in-text-out solutions may fail to leverage beneficial structural properties apparent to model stakeholders. Incorporating structures in the model development process requires a careful look at the problem setup, but often relatively straightforward implementation can pay significant dividends—a little structure goes a long way.

We conclude by positing that the next generation of AI systems will treat LLMs as powerful kernels upon which flexible inference procedures can be built to enhance complex reasoning. This approach, driven by the concept of inference-time compute, has the potential to significantly improve the problem-solving capabilities of AI.

# Acknowledgements

The list of people I want to thank is incredibly long, and despite my best efforts, it is certainly incomplete. Countless individuals have contributed to my PhD journey in big and small ways, and I am forever grateful for their support, guidance, and friendship.

First, thanks to Yiming for taking a chance on me and teaching me so much about research. Her guidance on writing, presenting ideas effectively, and asking insightful questions early on has been invaluable throughout my PhD journey. Graham has been an incredible mentor and pseudo-advisor to me. His ability to break down complex problems into manageable pieces has greatly influenced my research approach. I am grateful for his constant support and guidance, and for the countless hours, he spent discussing ideas and providing feedback on my work. Thanks to Niket for mentoring me from the early days of my PhD. Collaborating with Niket has been a truly enriching experience, and I have learned much from his expertise and insights. I am also thankful to Daniel for serving on my thesis committee and providing detailed comments on the thesis draft.

Uri taught me much about research, presenting ideas, leadership, and mentoring others. My PhD experience would have been significantly diminished without his guidance and the opportunity to collaborate with him. Shrimai and Dheeraj provided invaluable advice and support during the early years of my PhD. Amir taught me a lot about research, writing, and being diligent.

To my labmates - Donghan, Guokun, Jingzhou, Peter, Ruohong, Shanda, Shengyu, Yuexin, Zhiqing, and Zihang - thank you for creating a motivating lab environment that always pushed me to do more. Guokun deserves a special mention for sharing invaluable advice on navigating the PhD program, which likely increased my chances of success.

I am thankful for the great collaborations I have had with other students during my PhD. Working with Shuyan and Luyu was always an exciting and rewarding experience, and I looked forward to our discussions and the progress we made together. Yuwei was an excellent collaborator on KAIROS, and I appreciate his proactivity. Pranjal, Alex, Andrew, Syeda, Patrick, Pei, Amrith, and Tanmay: It was a pleasure collaborating with you.

I am grateful to Mausam, Sunita, and Soumen—my first research mentors—for inspiring me to pursue a PhD. Special thanks go to Rahul, who played a pivotal role in my decision to pursue a PhD. Thanks also to Mandar for helping with my applications and inspiring me to do more.

To my CMU friends - Prakhar, Sai, Sanket, Kundan, Adithya, Kaixin, and Torsten - thank you for making my time in Pittsburgh memorable. Adithya and Torsten, you were wonderful officemates and friends. Thanks to Kaixin for our helpful weekly meetings and his many useful suggestions. His early advice on structuring my thesis was particularly valuable.

I am deeply grateful for my friends who have been like family - Gary, Priyanka, Rahul, Remona, Veer, and Zuni. Their unwavering support throughout my PhD helped a lot.

Yifan has been by my side throughout my PhD journey. She has put up with my insane schedule and the demanding nature of the PhD program, always understanding and making everything easier for me. I want to thank my family in India for their understanding and support, even from a distance. Jagdish, my manager at Visa and Oracle, taught me many invaluable engineering and life skills that were very useful during my PhD. Finally, I would like to thank my friends Qiudi, Harshit, Rakshit, Rui, and Ankur for staying in contact and being supportive during the tiring times of my PhD.

I also want to thank the CMU staff, particularly Kate and Stacey, for their tireless work in supporting graduate students like myself.

# Contents


**1 Introduction** **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Thesis Overview . . . 6

**I Infusing Structure in Data for Finetuning** **8**

**2 Neural Language Modeling for Contextualized Temporal Graph Generation** **9**
- 2.1 Introduction . . . 9
- 2.2 Deriving Large-scale Dataset for the Temporal Graph Generation . . . 10
- 2.3 Model . . . 12
- 2.4 Experiments and Results . . . 14

**3 Conditional Set Generation with SEQ2SEQ models** **19**
- 3.1 Introduction . . . 19
- 3.2 Task . . . 21
- 3.3 Method . . . 21
- 3.4 Experiments . . . 25
- 3.5 Conclusion . . . 31

**4 Learning to Generate Performance Enhancing Code Edits** **32**
- 4.1 Introduction . . . 32
- 4.2 Performance Improving Edits (PIE) Dataset . . . 34
- 4.3 Learning to Improve Code Performance . . . 35
- 4.4 Appendix . . . 40

**II Structure-Assisted Modeling** **41**

**5 Politeness Transfer: A Tag and Generate Approach** **42**
- 5.1 Related Work . . . 43
- 5.2 Tasks and Datasets . . . 44
- 5.3 Methodology . . . 46
- 5.4 Experiments and Results . . . 50

**6 Think about it! Improving Defeasible Reasoning by First Modeling the Question Scenario** **55**
6.1 Introduction . . . 56
6.2 Task . . . 57
6.3 Approach . . . 57
6.4 Experiments . . . 62
6.5 Summary and Conclusion . . . 67

**III Leveraging Structure During Inference** **68**

**7 Language Models of Code are Few-Shot Commonsense Learners** **69**
7.1 Introduction . . . 69
7.2 GPT-2: Representing Commonsense structures with code . . . 72
7.3 Evaluation . . . 73
7.4 Analysis . . . 79

**8 PAL: Program-aided Language Models** **81**
8.1 Introduction . . . 81
8.2 Background: Few-shot Prompting . . . 82
8.3 Program-aided Language Models . . . 83
8.4 Experimental Setup . . . 84
8.5 Results . . . 86
8.6 Analysis . . . 87
8.7 Related Work . . . 88

**IV Post-Inference Enchancements for LLMs** **95**

**9 MemPrompt: Memory-assisted Prompt Editing with User Feedback** **96**
9.1 Introduction . . . 96
9.2 Approach . . . 98
9.3 Experiments . . . 104

**10 Self-Refine: Iterative Refinement with Self-Feedback** **110**
10.1 Introduction . . . 110
10.2 Iterative Refinement with SELF-REFINE . . . 111
10.3 Evaluation . . . 114
10.4 Analysis . . . 116
10.5 Related work . . . 120
10.6 Limitations and Discussion . . . 122
10.7 Conclusion . . . 122

**V A Case for Inference-time Compute: Conclusion and Future Work 123**
10.8 Enhancing LLM Problem-Solving with Inference-Time Computation . . . . . 125

**Bibliography 128**
Appendices . . . . . 165
A Chapter 2: Neural Language Modeling for Contextualized Temporal Graph Generation . . . . . 165
B Chapter 3: Conditional Set Generation with SEQ2SEQ models . . . . . 168
C Chapter 4: Learning Performance Improving Code Edits . . . . . 181
D Analysis of Generated Code Edits . . . . . 181
E Chapter 6: Think about it! Improving defeasible reasoning by first modeling the question scenario . . . . . 190
F Chapter: Politeness Transfer: A Tag and Generate Approach 5 . . . . . 203
G Chapter 7: Language Models of Code are Few-Shot Commonsense Learners . . . 203
H Chapter 8: Program Aided Language Models . . . . . 216
I Chapter 9: Memory-assisted Prompting . . . . . 237
J Chapter 10: Self-Refine . . . . . 253

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Next-token prediction is surprisingly expressive. In theory, a wide range of complex structures, including text, code, and proteins can be generated incrementally by generating one piece or *token* at a time.

It is therefore unsurprising that with the broad availability of user-friendly libraries for text-generation and reasoning, numerous tasks have been successfully framed within the seq2seq framework [Radford, 2018, Raffel et al., 2020a, Yangfeng Ji and Celikyilmaz, 2020]. This extends beyond tasks naturally suited to these paradigms, such as dialogue generation and summarization [Zhang et al., 2020b, Gehrmann et al., 2021b], to include tasks not traditionally associated with language models, like protein sequence prediction [Gómez-Bombarelli et al., 2018], graph generation [You et al., 2018], program synthesis [Nijkamp et al., 2022a, Chen et al., 2021b, Wang et al., 2021], and structured-commonsense reasoning [Bosselut et al., 2019].

While adapting tasks to fit existing tools is generally not recommended[1], the ease and accessibility of these libraries [Paszke et al., 2017, Wolf et al., 2019] can sometimes lead to overlooking the inherent trade-offs and limitations associated with using such out-of-the-box solutions. Often, developers only need to provide their input data in a prescribed format (e.g., a file of comma-separated input and output values), with the libraries handling the remaining steps. The simplicity of libraries facilitates quick implementation and experimentation. However, this convenience comes with a trade off.

### 1.1.1 Limitations of current LLM setups

In this thesis, we argue that recognizing and addressing these trade-offs is crucial for the practical and challenging deployments of text-generation and reasoning frameworks. These shortcomings include the brittleness of such frameworks in handling complex problems, the lack of mechanisms for receiving feedback, and their opaque, black-box nature [Ortega et al., 2021]. Our goal is to delve into these issues and explore potential solutions that can enhance the practicality and

[1] "*To a man with a hammer, everything looks like a nail.* - Mark Twain

robustness of text-generation and reasoning frameworks in real-world applications. We elaborate on these challenges next.

(1) **The ability to provide feedback**: Feedback is crucial for tailoring model outputs to user preferences and improving the overall user experience. However, current Seq2Seq models are not designed to receive direct feedback, making it challenging for users to influence or guide the model's output [Kreutzer et al., 2018, Jaques et al., 2019].

The ability to provide feedback would enable more interactive and user-driven outcomes, allowing for better customization and improved overall performance. For instance, in a dialogue system, a user looking for Italian restaurants in New York City might want to clarify or correct information provided by the Seq2Seq model. If the model suggests an incorrect location, there is no easy way for the user to give feedback and guide the model towards the desired answer. Worse, without an ability to retain the feedback, the model will continue to repeat the same mistake.

Several approaches have been proposed to address this issue, such as reinforcement learning from human feedback [Kreutzer et al., 2018, Jaques et al., 2019], actor-critic algorithms for sequence prediction [Bahdanau et al., 2017], and supervised learning [Stiennon et al., 2020, Ouyang et al., 2022b]. However, these methods often require additional training or substantial amounts of data, making them less suitable for few-shot learning or scenarios with limited data availability. Despite these advances, there remains a significant research gap in developing practical and efficient feedback mechanisms for Seq2Seq models in the context of few-shot learning. In this thesis, we aim to investigate this gap and explore novel methods that can effectively incorporate user feedback without the need for re-training, thereby enhancing the performance and adaptability of Seq2Seq models in real-world applications with limited data availability.

(2) **Brittleness due to mismatched representations**: A major challenge faced by Seq2Seq models is their brittleness when dealing with inputs or outputs that significantly deviate from the textual data they were trained on. This limitation can result in poor performance when applied to unconventional tasks or domains that require representations different from those encountered during training [Lake et al., 2017, Ratner et al., 2017]. Developing models capable of handling diverse and mismatched representations would not only improve their generalization capabilities but also expand their applicability to a broader range of tasks.

For example, a Seq2Seq model trained on a large corpus of English text might be ill-suited for handling input or output in a domain-specific language, such as mathematical equations or computer code. Addressing this gap in handling mismatched representations is essential for creating more versatile and robust Seq2Seq models that can adapt to a variety of real-world scenarios and tasks [Graber et al., 2018].

(3) **Failure to utilize structure inherent in the data**: A significant limitation of vanilla Seq2Seq models is their tendency to treat input and output data as unstructured sequences, often ignoring any underlying structure or patterns that could be exploited to enhance the model's understanding and generation capabilities [Bastings et al., 2017]. Incorporating domain-specific knowledge, structure, or constraints into the model architecture or training process would enable more accurate, efficient, and coherent output generation, leading to better performance in specialized tasks or domains.

**Key Capabilities for Human-Like Text Generation and Reasoning**

A common argument in favor of the simplicity of the next-token prediction objective is its perceived similarity to how humans process and generate language [Heilbron et al., 2022]. However, human reasoning exhibits nuances that current models struggle to replicate. A few examples highlight these limitations:

1. **Generating Multiple Candidates:** Humans often create and evaluate multiple options, a process not inherent in standard LLM outputs.
2. **Iterative Generation:** In tasks like writing, humans engage in an iterative process of review and refinement instead of one-shot generation done by LLMs.
3. **Contextual and World Knowledge:** Human communication relies on broader knowledge and contextual information beyond the immediate textual data.
4. **Tool Usage:** Humans use various tools to accomplish tasks. Crucially, humans realize when a specific tool is required.
5. **Question Reframing:** Humans often rephrase the question and retry.
6. **Prioritizing Simpler Tasks:** A common human problem-solving strategy involves tackling a problem's simpler parts first.

The examples share a common theme: the need to move beyond simplistic input/output relationships. LLMs offer remarkable capabilities, but to address the full spectrum of tasks, they need to be enhanced with more complex reasoning processes. This need is reflected in the rise of few-shot prompting techniques, where strategies like search, self-refinement, and tool usage are used to augment these models. Many of these techniques implicitly introduce elements of structure, explained next.

### 1.1.2 Infusing Structure: contribution of this thesis

Certain problems may offer an inherent structure that can be exploited for interpretability or effectiveness. For example, while solving commonsense reasoning questions, it may be useful to additionally condition the result on a knowledge graph that captures relevant relationships and dependencies [Han et al., 2020]. Addressing this gap and developing methods to incorporate structural information into Seq2Seq models has the potential to significantly improve their performance and applicability across a wide range of domains and tasks [Zhang et al., 2019a,c].

Structure is an ambiguous term with multiple interpretations within the field of AI [Newell et al., 1972, Russell, 2010]. For the purpose of this thesis, we adopt a broad perspective of structure that includes not only its use in organizing training data [Bengio et al., 2013, Schmidhuber, 2015], but also its role in the entire model development and deployment lifecycle, from enhancing training and inference outcomes [Vaswani et al., 2017, Devlin et al., 2019, Lake et al., 2017], to post-inference adjustments that increase the effectiveness of the final results [Nye et al., 2021b, Dohan et al., 2022].

**Definition 1** (Structure)**.** *In the context of Structure-Enhanced Generation and Reasoning, the term* ***structure*** *refers to:*

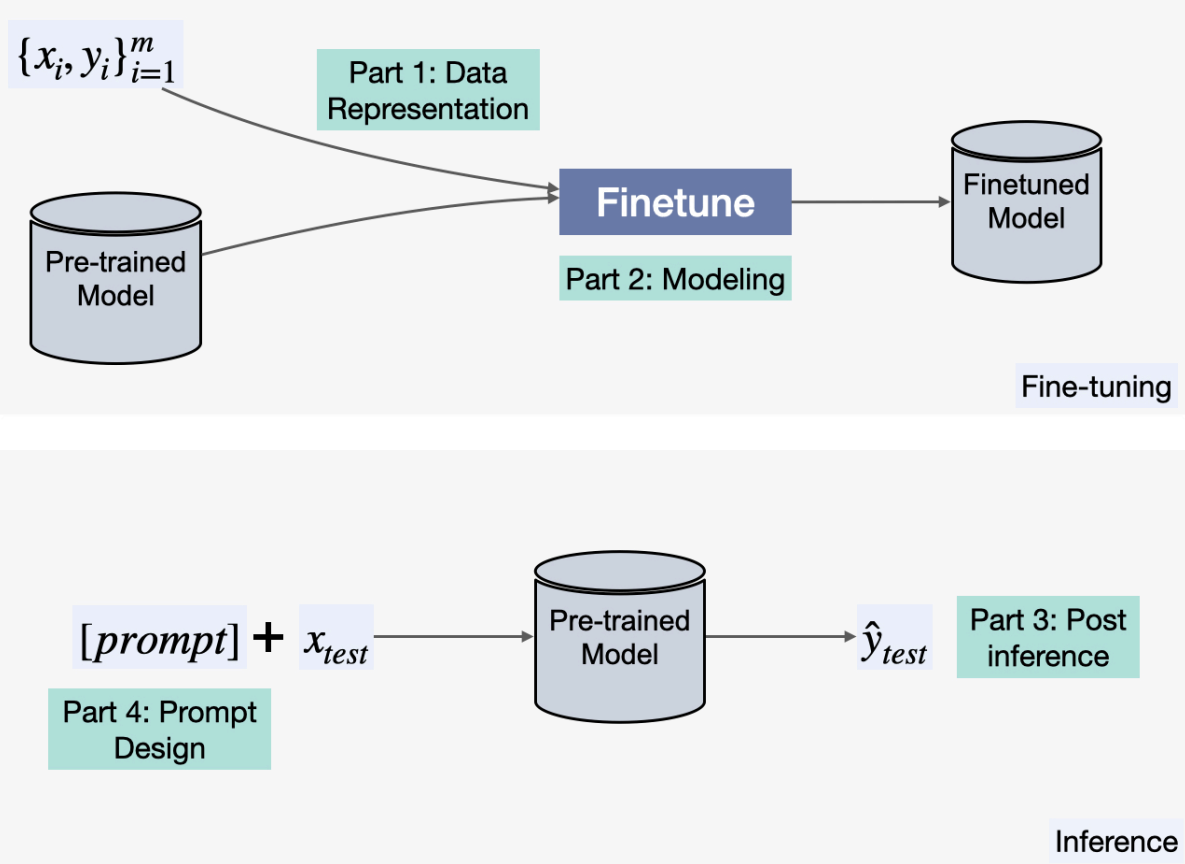
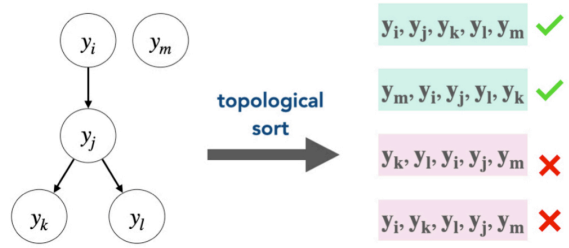



Figure 1.1: Overview of this thesis proposal: the goal of this thesis is to integrate structure in the model development and deployment pipeline.

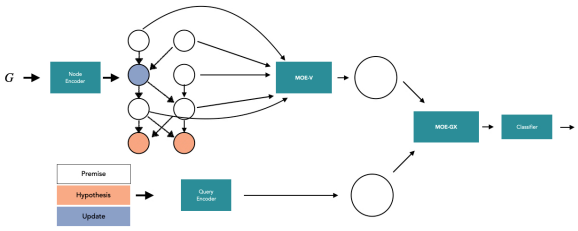
**a.** *Organization and representation of data, knowledge, or information in a systematic, hierarchical, or relational way [Pearl et al., 2000, Bengio et al., 2013, Hovy et al., 2013]. This helps capture the underlying relationships and dependencies between different elements, making it easier for AI systems to understand, generate, and reason with natural language. For example, organizing a knowledge graph to represent relationships between entities in a domain.*

**b.** *Leveraging the inherent structure present in the data or problem domain to optimize outcomes [Bahdanau et al., 2015a, Vaswani et al., 2017, Battaglia et al., 2018]. This involves using the structural properties of data or knowledge to improve reasoning, decision-making, or generation, as well as enhance the efficiency, interpretability, or scalability of AI systems. For example, using the structure of a parse tree to guide the generation of grammatically correct sentences.*

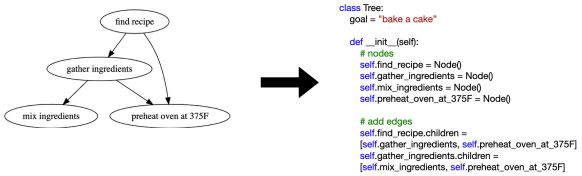
Note that this definition goes beyond the traditional definition of structure that focuses on the arrangement of data and includes the process in the definition. Thus, our definition of structure encompasses both the structuring of data and the process itself.



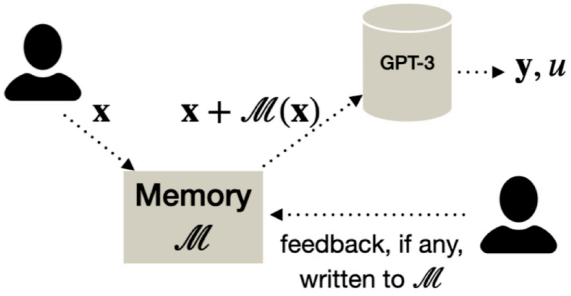

(a) Infusing Structure in Data (Part I)




(b) Structure-Enhanced Modeling (Part II)




(c) Exploiting Structure during Inference (Part III)




(d) Post-Inference LLM Enhancements (Part IV)

Figure 1.2: Examples from the four parts of the thesis: (a) Infusing Structure in Data for Finetuning, (b) Structure-Enhanced Modeling, (c) Exploiting Structure during Inference, and (d) Post-Inference Enhancements for LLMs.

## 1.2 Thesis Overview

This thesis investigates the significance of structure in contemporary language generation and reasoning models. The thesis is organized into four parts:

- **Part I: Infusing Structure in Data for Finetuning** covers three chapters that explore advanced applications of large language models (LLMs) in various tasks.
  - Chapter 2 examines event-level temporal graph generation for documents using LLMs (NAACL 2021). It presents the first study on using LLMs for automated generation of event-level temporal graphs for documents and demonstrates the effectiveness of the approach.
  - Chapter 3 introduces SETAUG, a novel algorithm for conditional set generation that effectively leverages order-invariance and cardinality properties (EMNLP 2022). By training sequence-to-sequence models on augmented data, this method achieves significant improvements across multiple benchmark datasets.
- **Part II: Structure-Assisted Modeling** delves into structure-enhanced generation and reasoning.
  - Chapter 5 focuses on text style transfer (ACL 2020) and proposes techniques for effective and interpretable style transfer without parallel data. The two-step process improves both performance and interpretability.
  - Chapter 6 investigates structured situational reasoning using graphs (ACL 2021, EMNLP 2021). It proposes a hierarchical mixture of experts model that learns to effectively condition on input noisy graphs for improved reasoning.
- Chapter 4 proposes a method for targeted algorithmic optimizations in programs using LLMs and a dataset of program trajectories (preliminary dataset verison accepted at DL4C 2022). This work is in progress and aims to improve the optimization process of programming tasks.
- **Part III: Leveraging Structure during Inference** explores approaches in graph generation, structured commonsense reasoning, and program-aided language models.
  - Chapter 7 introduces COCOGEN, a novel approach for structured commonsense reasoning using large language models (EMNLP 2022). It treats structured commonsense reasoning tasks as code generation tasks, allowing pre-trained LMs of code to perform better as structured commonsense reasoners.
  - Chapter 8 presents the Program-Aided Language models (PAL) approach, which leverages large language models for problem understanding and decomposition while outsourcing the solution step to a runtime (ICML 2023). This approach leads to improved performance in arithmetic and symbolic reasoning tasks.
- **Part IV: Post-Inference Enhancements for LLMs** examines two chapters focused on enhancing large language models (LLMs) through user interactions and iterative refinement.
  - Chapter 9 presents MEMPROMPT, an approach that pairs GPT-3 with a memory of user feedback for improved accuracy across diverse tasks (EMNLP 2022, NAACL 2022).

By pairing GPT-3 with a growing memory of recorded misunderstandings and user feedback for clarification, the system can generate enhanced prompts for new queries based on past user feedback. A variant of MEMPROMPT, called FB-NET, leverages feedback to fix mistakes in the outputs of a fine-tuned model for structured generation and was accepted at NAACL 2022.

- Chapter 10 introduces SELF-REFINE, a framework for iteratively refining LLM outputs by generating multi-aspect feedback, demonstrating significant improvements over direct generation in various tasks. The proposed work aims to extend Self-Refine by integrating planning approaches.

| Part | Work | Status |
|---|---|---|
| Part I | Event-level temporal graph generation | **NAACL 2021** |
| | Conditional set generation (SETAUG) | **EMNLP 2022** |
| | Performance-Improving Code Edits | **ICLR 2024** |
| Part II | TAGGEN | **ACL 2020** |
| | Graph-conditioned audio generation | **ASRU 2021** |
| | Structured situational reasoning | **ACL 2021, EMNLP 2021** |
| Part III | Graph generation (FLOWGEN) | **Dynn @ ICML 2022** |
| | Structured commonsense reasoning (COCOGEN) | **EMNLP 2022** |
| | Program-Aided Language models (PAL) | **ICML 2023** |
| Part IV | User feedback memory (MEMPROMPT, FB-NET) | **EMNLP 2022, NAACL 2022** |
| | Self-Refine | **Neurips 2023** |

Table 1.1: Thesis Status

# Part I

# Infusing Structure in Data for Finetuning

Before starting with the fine-tuning process, it is essential to leverage domain knowledge to introduce structure in the data used for fine-tuning. This approach requires minimal changes to the input data while retaining the original modeling process. However, these strategic modifications can significantly improve model performance. By incorporating inductive biases based on domain knowledge, which may not be inherently accessible to the model, we can enhance the usefulness of the same data during fine-tuning.

In this chapter, we discuss the following three works that exemplify the benefits of introducing structure in fine-tuning data using domain knowledge:

1. Contextualized Temporal Event Graphs: Converts the problem of temporal graph extraction (Chapter 2).
2. Conditional Set Generation using SEQ2SEQ models (Chapter 3).
3. Chapter 4 Large Pre-trained Language Models for Program Optimization: This chapter proposes to generate targeted edits to optimize programs algorithmically. The primary objective is to identify pairs of slow and fast programs, analyze their differences, and subsequently train an optimization model utilizing that information.

# Chapter 2

# Neural Language Modeling for Contextualized Temporal Graph Generation

## 2.1 Introduction

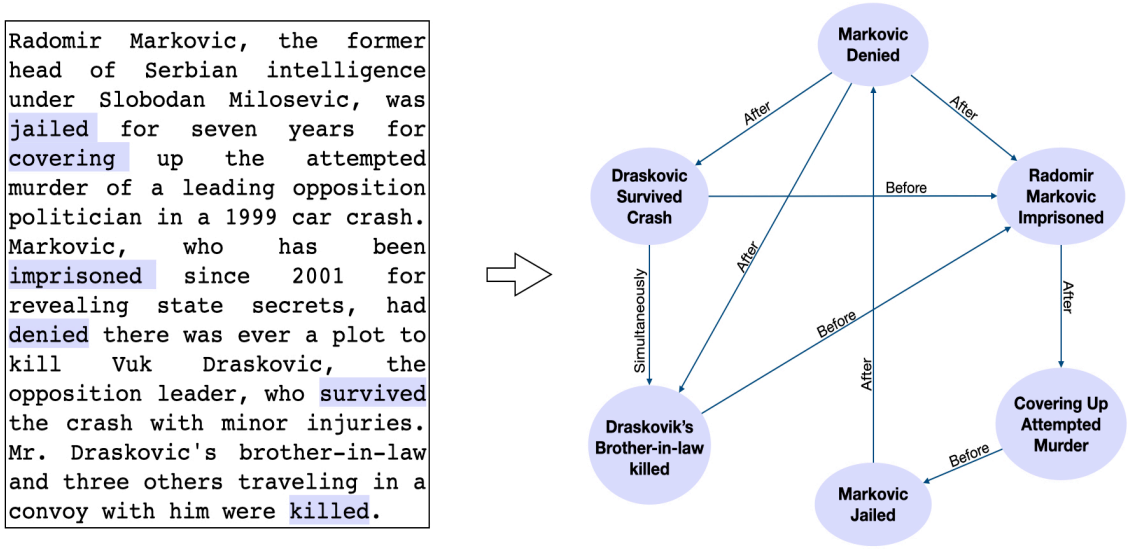



Figure 2.1: Task overview: given a document (left), automatically extract a temporal graph (right).

Temporal reasoning is crucial for analyzing the interactions among complex events and producing coherent interpretations of text data Duran et al. [2007]. There is a rich body of research on the use of temporal information in a variety of important application domains, including topic detection and tracking Makkonen et al. [2003], information extraction Ling and Weld [2010], parsing of clinical records Lin et al. [2016], discourse analysis Evers-Vermeul et al. [2017], and question answering Ning et al. [2020].

Graphs are a natural choice for representing the temporal ordering among events, where the nodes are the individual events, and the edges capture temporal relationships such as "before", "after" or "simultaneous". Representative work on automated extraction of such graphs from textual documents includes the early work by Chambers and Jurafsky [2009], where the focus is on the construction of event chains from a collection of documents, and the more recent CAEVO Chambers et al. [2014] and Cogcomptime Ning et al. [2018], which extract a graph for each input document instead. These methods focus on rule-based and statistical sub-modules to extract verb-centered events and the temporal relations among them.

As an emerging area of NLP, large scale pre-trained language models have made strides in addressing challenging tasks like commonsense knowledge graph completion Bosselut et al. [2019] and task-oriented dialog generation Budzianowski and Vulić [2019]. These systems typically fine-tune large language models on a corpus of a task-specific dataset. However, these techniques have not been investigated for temporal graph extraction.

This work focuses on the problem of generation of an event-level temporal graph for each document, and we refer to this task as *contextualized* graph generation. We address this open challenge by proposing a novel reformulation of the task as a sequence-to-sequence mapping problem Sutskever et al. [2014], which enables us to leverage large pre-trained models for our task. Further, different from existing methods, our proposed approach is completely end-to-end and eliminates the need for a pipeline of sub-systems commonly used by traditional methods.

We also address a related open challenge, which is a prerequisite to our main goal: the difficulty of obtaining a large quantity of training graphs with human-annotated events and temporal relations. To this end, we automatically produce a large collection of document-graph pairs by using CAEVO [Chambers et al., 2014], followed by a few rule-based post-processing steps for pruning and noise reduction. We then encode the graph in each training pair as a string in the graph representation format DOT, transforming the text-to-graph mapping into sequence-to-sequence mapping. We fine-tune GPT-2 on this dataset of document-graph pairs, which yields large performance gains over strong baselines on system generated test set and closely matches or even outperforms CAEVO on TimeBank-Dense Cassidy et al. [2014] on multiple metrics. Figure 1 shows an example of the input document and the generated graph by our system.

As an additional contribution, this work demonstrates the feasibility of knowledge distillation from complex, multi-step tools. By generating a dataset using a tool like CAEVO and fine-tuning a language model on it, we successfully distill knowledge about event identification and temporal relation extraction. The model effectively combines its general world understanding gained during pre-training with the task-specific knowledge encoded by the traditional tool. This approach has the potential to be broadly applicable in other domains where complex tools can be used to create datasets for fine-tuning large language models.

## 2.2 Deriving Large-scale Dataset for the Temporal Graph Generation

**Definitions and Notations:** Let $\mathbf{G}(\mathbf{V}, \mathbf{E})$ be a temporal graph associated with a document $\mathbf{D}$, such that vertices $\mathbf{V}$ are the events in document $\mathbf{D}$, and the edges $\mathbf{E}$ are temporal relations (links)

between the events. Every temporal link in $\mathbf{E}$ takes the form $r(e_q, e_t)$ where the query event $e_q$ and the target event $e_t$ are in $\mathbf{V}$, and $r$ is a temporal relation (e.g., before or after). In this work, we undertake two related tasks of increasing complexity: i) Node generation, and ii) Temporal graph generation:

**Task 1: Node Generation:** *Let $r(e_q, e_t)$ be an edge in $\mathbf{E}$. Let $C_r$ be the set of sentences in the document $\mathbf{D}$ that contains the events $e_q$ or $e_t$ or are adjacent to them. Given a query consisting of $C_r$, $r$, and $e_q$, generate $e_t$.*

**Task 2: Temporal Graph Generation:** *Given a document $\mathbf{D}$, generate the corresponding temporal graph $\mathbf{G}(\mathbf{E}, \mathbf{V})$.*

Figure 2.1 illustrates the two tasks. Task 1 is similar to knowledge base completion, except that the output events $e_q$ are generated, and not drawn from a fixed set of events. Task 2 is significantly more challenging, requiring the generation of both the structure and semantics of $\mathbf{G}$.

The training data for both the tasks consists of tuples $\{(x_i, y_i)\}_{i=1}^{N}$. For Task 1, $x_i$ is the concatenation of the query tokens $(C_r, e_q, r)$, and $y_i$ consists of tokens of event $e_t$. For Task 2, $x_i$ is the $i^{th}$ document $\mathbf{D_i}$, and $y_i$ is the corresponding temporal graph $\mathbf{G_i}$.

We use the New York Times (NYT) Annotated Corpus [1] to derive our dataset of document-graph pairs. The corpus has 1.8 million articles written and published by NYT between 1987 and 2007. Each article is annotated with a hand-assigned list of descriptive terms capturing its subject(s). We filter articles with one of the following descriptors: {"bomb", "terrorism", "murder", "riots", "hijacking", "assassination", "kidnapping", "arson", "vandalism", "hate crime", "serial murder", "manslaughter", "extortion"}, yielding 89,597 articles, with a total of 2.6 million sentences and 66 million tokens. For each document $\mathbf{D}$, we use CAEVO Chambers et al. [2014] to extract the dense temporal graph consisting of i) the set of verbs, and ii) the set of temporal relations between the extracted verbs. CAEVO extracts six temporal relations: before, after, includes, is included, simultaneous, and vague.

**Datasets for Task 1 and Task 2** After running the pruning and clustering operations outlined above on 89k documents, we obtain a corpus of over 890,677 text-graph pairs, with an average of 120.31 tokens per document, and 3.33 events and 4.91 edges per graph. These text-graph pairs constitute the training data for Task 2. We derive the data for Task 1 from the original (undivided) 89k graphs (each document-graph pair contributes multiple examples for Task 1). In Task 1 data, nearly 80% of the queries $(C_r, e_q, r)$ had a unique answer $e_t$, and nearly 16% of the queries had two different true $e_t$. We retain examples with multiple true $e_t$ in the training data because they help the model learn diverse temporal patterns that connect two events. For fairness, we retain such cases in the test set. Table 2.1 lists the statistics of the dataset. The splits were created using non-overlapping documents.

### 2.2.1 Graph Representation

We use language models to generate each graph as a sequence of tokens conditioned on the document, thus requiring that the graphs are represented as strings. We use DOT language Gansner et al. [2006] to format each graph as a string. While our method does not rely on any specific graph representation format, we use DOT as it supports a wide variety of graphs and allows

[1] https://catalog.ldc.upenn.edu/LDC2008T19

| Task | train | valid | test |
|---|---|---|---|
| Task 1 | 4.26 | 0.54 | 0.54 |
| Task 2 | 0.71 | 0.09 | 0.09 |

Table 2.1: Dataset statistics (counts in million).

augmenting graphs with node, edge, and graph level information. Further, graphs represented in DOT are readily consumed by popular graph libraries like NetworkX Hagberg et al. [2008b], making it possible to use the graphs for several downstream applications. Figure 2.2 shows an example graph and the corresponding DOT code. The edges are listed in the order in which their constituent nodes appear in the document. This design choice was inspired by our finding that a vast majority of temporal links exist between events that are either in the same or in the adjoining sentence (this phenomenon was also observed by Ning et al. [2017]). Thus, listing the edges in the order in which they appear in the document adds a simple inductive bias of locality for the auto-regressive attention mechanism, whereby the attention weights *slide* from left to right as the graph generation proceeds. Additionally, a fixed order makes the problem well defined, as the mapping between a document and a graph becomes deterministic.

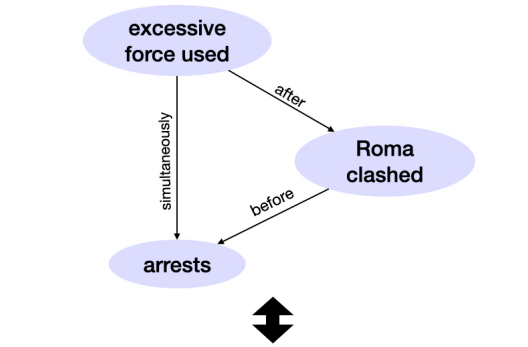



Figure 2.2: Temporal graph and the corresponding DOT representation for the sentence: *Roma clashed fiercely with the police, leading to arrests in which Roma activists said excessive force was used.*

## 2.3 Model

The training data $\mathbf{X}$ for both Tasks 1 and 2 comprises of tuples $\{(\S x_i, \S y_i)\}_{i=1}^{N}$. For task 1 (node generation), $\S x_i$ the concatenation of context, the source, node, and the relation. The target $\S y_i$ consists of the tokens of the target event. For task 2 (graph generation), $\S x_i$ is a document and $\S y_i$ is the corresponding temporal graph represented in DOT. We train a (separate) conditional language

| Method | Dataset | BLEU | MTR | RG | ACC |
|---|---|---|---|---|---|
| SEQ2SEQ | TG-Gen (-C) | 20.20 | 14.62 | 31.95 | 19.68 |
| SEQ2SEQ | TG-Gen | 21.23 | 16.48 | 35.54 | 20.99 |
| GPT-2 | TG-Gen (-C) | 36.60 | 25.11 | 43.07 | 35.07 |
| GPT-2 | TG-Gen | **62.53** | **43.78** | **69.10** | **61.35** |
| SEQ2SEQ | TB-Dense (-C) | 11.55 | 9.23 | 21.87 | 10.06 |
| SEQ2SEQ | TB-Dense | 16.68 | 12.69 | 27.75 | 13.97 |
| GPT-2 | TB-Dense (-C) | 22.35 | 15.04 | 27.73 | 20.81 |
| GPT-2 | TB-Dense | **52.21** | **35.69** | **57.98** | **47.91** |

Table 2.2: Node Generation (task 1) results.

model to solve both the tasks. Specifically, given a training corpus of the form $\{(\S x_i, \S y_i)\}$, we aim to estimate the distribution $p_\theta(\S y_i \mid \S x_i)$. Given a training example $(\S x_i, \S y_i)$ we set $\S u_i = \S x_i \| \S y_i$[2]. $p_\theta(\S u_i)$ can then be factorized as a sequence of auto-regressive conditional probabilities using the chain rule: $p_\theta(\S u_i) = \prod_{k=1}^{n} p(u_{i,k} | \S u_{i,<k})$, where $u_{i,k}$ denotes the $k^{th}$ token of the $i^{th}$ sequence, and $\S u_{i,<k}$ denotes the sequence of tokens $\{u_1, u_2, ..., u_{k-1}\}$. Language models are typically trained by minimizing a cross-entropy loss $-\log p_\theta(\S u_i)$ over each sequence $\S u_i$ in $\mathbf{X}$. However, the cross-entropy loss captures the joint distribution $p_\theta(\S x_i, \S y_i)$, and is not aligned with our goal of learning conditional distribution $p_\theta(\S y_i \mid \S x_i)$. To circumvent this, we train our model by masking the loss terms corresponding to the input $\S x_i$, similar to Bosselut et al. [2019]. Let $\S m_i$ be a mask vector for each sequence $\S u_i$, set to 0 for positions corresponding to $\S x_i$, and 1 otherwise i.e. $m_{i,j} = 1$ if $j > |\S x_i|$, else 0. We combine the mask vector with our factorization of $p_\theta(\S u_i)$ to formulate a *masked* language modeling loss $\mathcal{L}$, which is minimized over the training corpus $\mathbf{X}$ to estimate the optimal $\theta$:

$$\mathcal{L}(\mathbf{X}) = -\sum_{i=1}^{|\mathbf{X}|} \sum_{j=1}^{|x_i|+|y_i|} m_{i,j} * \log\left(p_\theta\left(u_{i,j} | \S u_{i,<j}\right)\right)$$

Note that the formulation of masked loss is opaque to the underlying architecture, and can be implemented with a simple change to the loss function. In practice, we use GPT-2 Radford et al. [2019] based on transformer architecture Vaswani et al. [2017] for our implementation. Having trained a $p_\theta$ for each task, we generate a node $(\S y)$ given a query $(\S x)$ (for Task 1), or a graph $(\S y)$ given a document $(\S x)$ (for Task 2) by drawing samples from the appropriate $p_\theta(\S y \mid \S x)$ using nucleus sampling Holtzman et al. [2020]. We provide more details of our training procedure and the architecture in the Appendix (A.3).

## 2.4 Experiments and Results

### 2.4.1 Evaluation Datasets

We evaluate our method on two different datasets: i) **TG-Gen**: Test split of synthetically created dataset (Section 2.2), and ii) **TB-Dense:** A mixed-domain corpus, with human-annotated temporal annotations. We create TB-Dense from the test splits of TimeBank-Dense Cassidy et al. [2014] by applying the same pre-processing operations as we did for TG-Gen. TB-Dense forms a very challenging dataset for our task because of domain mismatch; our system was trained on a corpus of terrorism-related events, whereas TB-Dense includes documents from a wide array of domains, forming a zero-shot evaluation scenario for our method.

**SEQ2SEQ:** We train a bi-directional LSTM Hochreiter and Schmidhuber [1997] based sequence-to-sequence model Bahdanau et al. [2015b] with global attention Luong et al. [2015] and a hidden size of 500 as a baseline to contrast with GPT-2. The token embeddings initialized using 300-dimensional pre-trained Glove Pennington et al. [2014].

### 2.4.2 Task 1: Node Generation

| |
|---|
| **Paragraph:** *Mr. Grier, a former defensive lineman for the New York Giants who was **ordained** as a minister in 1986, **testified** on Dec. 9 that he had **visited** Mr. Simpson a month earlier* |

Table 2.3: An example of GPT-2 *fixing* the label given by CAEVO. Given a query *event **after** "Mr. Grier visited"*, CAEVO incorrectly extracts *Mr. Grier ordained*, whereas GPT-2 generates the correct event: *Mr. Grier testified*.

**Metrics** Given a query $(C_r, e_q, r)$, with $C_r$ being the context (sentences containing events $e_q, e_t$ and their neighboring sentences) and $e_q$ as the source event, Task 1 is to generate a target event $e_t$ such that $r(e_q, e_t)$. We format each query as "In the context of $C$, what happens $r$ $e_q$?". We found formatting the query in natural language to be empirically better. Let $\hat{e_t}$ be the system generated event. We compare $e_t$ vs. $\hat{e_t}$ using BLEU Papineni et al. [2002], METEOR Denkowski and Lavie [2011], and ROUGE Lin [2004][3], and measure the accuracy (ACC) as the fraction of examples where $e_t = \hat{e_t}$.

**Results on TG-Gen** The results are listed in Table 2.2. Unsurprisingly, GPT-2 achieves high scores across the metrics showing that it is highly effective in generating correct events. To test the generative capabilities of the models, we perform an ablation by removing the sentence containing the target event $e_t$ from $C_r$ (indicated with -*C*). Removal of context causes a drop in performance for both GPT-2 and SEQ2SEQ, showing that it is crucial for generating temporal

[2] || denotes concatenation

[3] Sharma et al. [2017], https://github.com/Maluuba/nlg-eval

events. However, GPT-2 obtains higher relative gains with context present, indicating that it uses its large architecture and pre-training to use the context more efficiently. GPT-2 also fares better as compared with SEQ2SEQ in terms of drop in performance for the out-of-domain TB-Dense dataset on metrics like accuracy ($-21\%$ vs. $-33\%$) and BLEU ($-16\%$ vs. $-21\%$), indicating that pre-training makes helps GPT-2 in generalizing across the domains.

**Human Evaluation** To understand the nature of errors, we analyzed 100 randomly sampled incorrect generations. For 53% of the errors, GPT-2 generated a non-salient event which nevertheless had the correct temporal relation with the query. Interestingly, for 10% of the events, we found that GPT-2 *fixed* the label assigned by CAEVO (Table 2.3), i.e., $e_t$ was incorrect but $\hat{e}_t$ was correct.

### 2.4.3 Task 2: Graph Generation

| | Dataset | BLEU | MTR | RG | DOT% |
|---|---|---|---|---|---|
| SEQ2SEQ | TG-Gen | 4.79 | 15.03 | 45.95 | 86.93 |
| GPT-2 | TG-Gen | **37.77** | **37.22** | **64.24** | **94.47** |
| SEQ2SEQ | TB-Dense | 2.61 | 12.76 | 28.36 | 89.31 |
| GPT-2 | TB-Dense | **26.61** | **29.49** | **49.26** | **92.37** |

Table 2.4: Graph string metrics.

| | Dataset | $v_P$ | $v_R$ | $v_{F_1}$ | $e_P$ | $e_R$ | $e_{F_1}$ |
|---|---|---|---|---|---|---|---|
| SEQ2SEQ | TG-Gen | 36.84 | 24.89 | 28.11 | 9.65 | 4.29 | 4.70 |
| GPT-2 | TG-Gen | **69.31** | **66.12** | **66.34** | **27.95** | **25.89** | **25.22** |
| SEQ2SEQ | TB-Dense | 24.86 | 15.25 | 17.99 | 4.7 | 0.14 | 0.24 |
| CAEVO | TB-Dense | 37.53 | **79.83** | **48.96** | 7.95 | **14.62** | **8.96** |
| GPT-2 | TB-Dense | **45.96** | 48.44 | 44.97 | **8.74** | 8.89 | 7.96 |

Table 2.5: Graph semantic metrics.

**Metrics** Let $\mathbf{G_i}(\mathbf{V_i}, \mathbf{E_i})$ and $\hat{\mathbf{G}}_\mathbf{i}(\hat{\mathbf{V}}_\mathbf{i}, \hat{\mathbf{E}}_\mathbf{i})$ be the true and the generated graphs for an example $i$ in the test corpus. Please recall that our proposed method generates a graph from a given document as a string in DOT. Let $\S y_i$ and $\S \hat{y}_i$ be the string representations of the true and generated graphs. We evaluate our generated graphs using three types of metrics:

1. **Graph string metrics**: To compare $\S y_i$ vs. $\S \hat{y}_i$, we use BLEU, METEOR, and ROUGE, and also measure parse accuracy (DOT%) as the % of generated graphs $\S \hat{y}_i$ which are valid DOT files.

2. **Graph structure metrics** To compare the structures of the graphs $\mathbf{G_i}$ vs. $\hat{\mathbf{G}}_\mathbf{i}$, we use i) Graph edit distance (GED) Abu-Aisheh et al. [2015] - the minimum numbers of edits required to transform the predicted graph to the true graph by addition/removal of an edge/node; ii) Graph isomorphism (ISO) Cordella et al. [2001] - a binary measure set to 1 if the graphs are isomorphic (without considering the node or edge attributes); iii) The average graph size ($|\mathbf{V_i}|, |\mathbf{E_i}|, |\hat{\mathbf{V}}_\mathbf{i}|, |\hat{\mathbf{E}}_\mathbf{i}|$) and the average degree ($\mathrm{d}(\mathbf{V})$).

3. **Graph semantic metrics:** We evaluate the node sets ($\mathbf{V_i}$ vs. $\hat{\mathbf{V}}_\mathbf{i}$) and the edge sets ($\mathbf{E_i}$ vs. $\hat{\mathbf{E}}_\mathbf{i}$) to compare the semantics of the true and generated graphs. For every example $i$, we calculate node-set precision, recall, and $F_1$ score, and average them over the test set to obtain node precision ($v_P$), recall ($v_R$), and $F_1$ ($v_F$). We evaluate the predicted edge set using temporal awareness UzZaman and Allen [2012], UzZaman et al. [2013]. For an example $i$, we calculate $e^i_P = \frac{|\hat{\mathbf{E}}_\mathbf{i}^- \cap \mathbf{E_i}^+|}{|\hat{\mathbf{E}}_\mathbf{i}^-|}$, $e^i_R = \frac{|\hat{\mathbf{E}}_\mathbf{i}^+ \cap \mathbf{E_i}^-|}{|\mathbf{E_i}^-|}$ where symbol $+$ denotes the temporal transitive closure Allen [1983] of the edge set. Similarly, $-$ indicates the reduced edge set, obtained by removing all the edges that can be inferred from other edges transitively. The $F_1$ score $e^i_{F_1}$ is the harmonic mean of $e^i_P$ and $e^i_R$, and these metrics are averaged over the test set to obtain the temporal awareness precision ($e_P$), recall ($e_R$), and $F_1$ score ($e_{F_1}$). Intuitively, the node metrics judge the quality of generated events in the graph, and the edge metrics evaluate the corresponding temporal relations.

**Results** Tables 2.4, 2.6, and 2.5 present results for graph generation, and we discuss them next.

| | Dataset | $\lvert\mathbf{V}\rvert$ | $\lvert\mathbf{E}\rvert$ | $d(\mathbf{V})$ | GED ↓ | ISO ↑ |
|---|---|---|---|---|---|---|
| True | TG-Gen | 4.15 | 5.47 | 1.54 | 0 | 100 |
| SEQ2SEQ | TG-Gen | 2.24 | 2.23 | 1.12 | 6.09 | 32.49 |
| GPT-2 | TG-Gen | **3.81** | **4.60** | **1.40** | **2.62** | **41.66** |
| True | TB-Dense | 4.39 | 6.12 | 2.02 | 0 | 100 |
| SEQ2SEQ | TB-Dense | 2.21 | 2.20 | 1.11 | 6.22 | 23.08 |
| CAEVO | TB-Dense | 10.73 | 17.68 | 2.76 | 18.68 | 11.11 |
| GPT-2 | TB-Dense | **3.72** | **4.65** | **1.75** | **4.05** | **24.00** |

Table 2.6: Graph structure metrics.

**GPT-2 vs. SEQ2SEQ** GPT-2 outperforms SEQ2SEQ on all the metrics by a large margin in both fine-tuned (TG-Gen) and zero-shot settings (TB-Dense). GPT-2 generated graphs are closer to the true graphs in size and topology, as shown by lower edit distance and a higher rate of isomorphism in Table 2.6. Both the systems achieve high parsing rates (DOT %), with GPT-2 generating valid DOT files 94.6% of the time. The high parsing rates are expected, as even simpler architectures like vanilla RNNs have been shown to generate syntactically valid complex structures like LaTeXdocuments with ease Karpathy [2015].

**GPT-2 vs. CAEVO** We compare the graphs generated by GPT-2 with those extracted by CAEVO Chambers et al. [2014][4] from the TB-Dense documents. We remove all the vague edges and the light verbs from the output of CAEVO for a fair comparison. Please recall that CAEVO is the tool we used for creating the training data for our method. Further, CAEVO was trained using TB-Dense, while GPT-2 was not. Thus, CAEVO forms an upper bound over the performance of GPT-2. The results in Tables 2.5 and 2.6 show that despite these challenges, GPT-2 performs strongly across a wide range of metrics, including GED, ISO, and temporal awareness. Comparing the node-set metrics,

[4] https://github.com/nchambers/caevo

| |
|---|
| **Top 10 Verbs:** found, killed, began, called, want, took, came, used, trying, asked<br>**Randomly Sampled Verbs:** shooting, caused, accused, took, conceived, visit, vowing, play, withdraw, seems |

Table 2.7: Verbs in GPT-2 generated graphs.

| Query $(C, e_q, r)$ | $e_t$ | Explanation |
|---|---|---|
| The suspected car bombings...turning busy streets...Which event happened before the suspected car bombings? | many cars drove | *Plausible:* The passage mentions busy streets and car bombing. |
| He...charged...killed one person. Which event happened after he was charged? | He was acquitted | *Somewhat plausible:* An acquittal is a possible outcome of a trial. |

Table 2.8: Sample open-ended questions and the answers $e_t$ generated by our system. Note that the answers generated by our system $e_t$ are complete event phrases (not just verbs).

we see that GPT-2 leads CAEVO by over eight precision points ($v_P$), but loses on recall ($v_R$) as CAEVO extracts nearly every verb in the document as a potential event. On temporal awareness (edge-metrics), GPT-2 outperforms both CAEVO and SEQ2SEQ in terms of average precision score $e_P$ and achieves a competitive $e_{F_1}$ score. These results have an important implication: they show that our method can best or match a pipeline of specialized systems given reasonable amounts of training data for temporal graph extraction. CAEVO involves several sub-modules to perform part-of-speech tagging, dependency parsing, event extraction, and several statistical and rule-based systems for temporal extraction. In contrast, our method involves no hand-curated features, is trained end-to-end (single GPT-2), and can be easily scaled to new datasets.

**Node extraction and Edge Extraction** The node-set metrics in Table 2.5 shows that GPT-2 avoids generating noisy events (high $P$), and extracts salient events (high $R$). This is confirmed by manual analysis, done by randomly sampling 100 graphs from the GPT-2 generated graphs and isolating the main verb in each node (Table 2.7). We provide several examples of generated graphs in the Appendix. We note from Table 2.5 that the relative difference between the $e_{F_1}$ scores for GPT-2 and SEQ2SEQ (25.22 vs. 4.70) is larger than the relative difference between their $v_{F_1}$ scores (66.34 vs. 28.11), showing that edge-extraction is the more challenging task which allows GPT-2 to take full advantage of its powerful architecture. We also observe that edge extraction ($e_{F_1}$) is highly sensitive to node extraction ($v_{F_1}$); for GPT-2, a 27% drop in $v_{F_1}$ (66.34 on TG-Gen vs. 44.97 on TB-Dense) causes a 68% drop in $e_{F_1}$ (25.22 on TG-Gen vs. 7.96 on TB-Dense). As each node is connected to multiple edges on average (Table 2.6), missing a node during the generation process might lead to multiple edges being omitted, thus affecting edge extraction metrics disproportionately.

### 2.4.4 Answering for Open-ended Questions

A benefit of our approach of using a pre-trained language model is that it can be used to *generate* an answer for open-ended temporal questions. Recently, Ning et al. [2020] introduced Torque, a temporal reading-comprehension dataset. Several questions in Torque have no answers, as they concern a time scope not covered by the passage (the question is about events not mentioned in the passage). We test the ability of our system for generating plausible answers for such questions out of the box (i.e., without training on Torque). Given a (passage, question) pair, we create a query $(C, e_q, r)$, where $C$ is the passage, and $e_q$ and $r$ are the query event and temporal relation in the question. We then use our GPT-2 based model for node-generation trained without context and generate an answer $e_t$ for the given query. A human-judge rated the answers generated for 100 such questions for plausibility, rating each answer as being *plausible*, *somewhat plausible*, or *incorrect*. For each answer rated as either *plausible* or *somewhat plausible*, the human-judge wrote a short explanation to provide a rationale for the plausibility of the generated event. Out of the 100 questions, the human-judge rated 22 of the generated answers as plausible and ten as somewhat plausible, showing the promise of our method on this challenging task (Table 2.8).

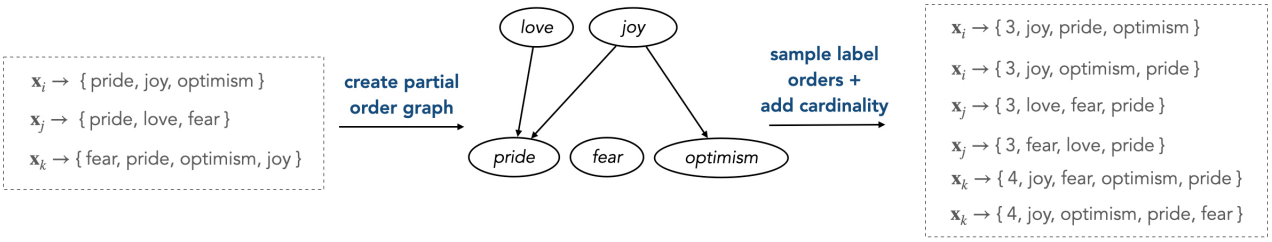



Figure 3.1: An illustrative task where given an input $\boldsymbol{x}$, the output is a set of emotions. Our method first discovers a partial order graph (middle) in which specific labels (joy) come before more general labels (pride). Listing the specific labels first gives the model more clues about the rest of the set. Topological samples from this partial order graph are label sequences that can be efficiently generated using SEQ2SEQ models. The size of each set is also added as the first element for joint modeling of output with size.

# Chapter 3

# Conditional Set Generation with SEQ2SEQ models

## 3.1 Introduction

Conditional set generation is the task of modeling the distribution of an output set given an input sequence of tokens [Kosiorek et al., 2020]. Several NLP tasks are instances of set generation, including open-entity typing [Choi et al., 2018, Dai et al., 2021], fine-grained emotion classification [Demszky et al., 2020], and keyphrase generation [Meng et al., 2017, Yuan et al., 2020, Ye et al., 2021]. The recent successes of the pretraining-finetuning paradigm have encouraged a formulation of set generation as a SEQ2SEQ generation task [Vinyals et al., 2016, Yang et al., 2018a, Meng et al., 2019, Ju et al., 2020].

In this paper, we posit that modeling set generation as a vanilla SEQ2SEQ generation task is sub-optimal, because the SEQ2SEQ formulations do not explicitly account for two key properties of a set output: *order-invariance* and *cardinality*. Forgoing order-invariance, vanilla SEQ2SEQ generation treats a set as a sequence, assuming an arbitrary order between the elements it outputs. Similarly, the cardinality of sets is ignored, as the number of elements to be generated is typically

not modeled.

Prior work has highlighted the importance of these two properties for set output through loss functions that encourage order invariance [Ye et al., 2021], exhaustive search over the label space for finding an optimal order [Qin et al., 2019, Rezatofighi et al., 2018, Vinyals et al., 2016], and post-processing the output [Nag Chowdhury et al., 2016]. Despite the progress, several important gaps remain. First, exhaustive search does not scale with large output spaces typically found in NLP problems, thus stressing the need for an optimal sampling strategy for the labels. Second, cardinality is still not explicitly modeled in the SEQ2SEQ setting despite being an essential aspect for a set. Finally, architectural modifications required for specialized set-generation techniques might not be viable for modern large-language models.

We address these challenges with a novel data augmentation strategy. Specifically, we take advantage of the auto-regressive factorization used by SEQ2SEQ models and (i) impose an *informative* order over the label space, and (ii) explicitly model *cardinality*. First, the label sets are converted to sequences using informative orders by grouping labels and leveraging their dependency structure. Our method imposes a partial order graph over the labels to efficiently search for such informative orders over a combinatorial space, where the nodes are the labels, and the edges denote the conditional dependence relations. We then generate the training data with orders over the label set that are sampled by performing topological traversals over the graph. Labels that are not constrained by dependency relations are augmented in different positions in each sample, reinforcing the order-invariance. We then create an augmented training dataset, where each input instance is paired with various valid label sequences sampled from the dependency graph. Next, we jointly model a set with its cardinality by simply prepending the set size to the output sequence. This strategy aligns with the current trend of very large language models which do not lend themselves to architectural modifications but increasingly rely on the informativeness of the inputs [Yang et al., 2020, Liu et al., 2021a].

Figure 3.1 illustrates the key intuitions behind our method using sample task where given an input $\boldsymbol{x}$ (say a conversation), the output is a set of emotions ($\mathbb{Y}$). While the original data may contain various orderings of emotions, we discover that certain orderings are generally more informative. Consider a case where one of the emotions is *joy*, which leads to a more general emotion of *pride*. After first generating *joy*, the model can generate *pride* with certainty (*joy* leads to *pride* in all samples). In contrast, the reverse order (generating *pride* first) still leaves room for multiple possible emotions (joy and love). The order $[\text{joy}, \text{pride}]$ is thus more informative than $[\text{pride}, \text{joy}]$. The cardinality of a set can also be helpful. In our example, joy contains two sub-emotions, and love contains one. A model that first predicts the number of sub-emotions can be more precise and avoid over-generation, a significant challenge with language generation models [Welleck et al., 2020, Fu et al., 2021]. We efficiently sample such informative orders from the combinatorial space of all possible orders and jointly model cardinality by leveraging the auto-regressive nature of SEQ2SEQ models.

**Our contributions**

(i) We show an efficient way to model sequence-to-set prediction as a SEQ2SEQ task by jointly modeling the cardinality and augmenting the training data with informative sequences using our novel TSAMPLE data augmentation approach. (§3.3.1, 3.3.2).

(ii) We theoretically ground our approach: treating the order as a latent variable, we show that our method serves as a better proposal distribution in a variational inference framework. (§3.3.1)
(iii) With our approach, SEQ2SEQ models of different sizes achieve a ∼20% relative improvement on four real-world tasks, with no additional annotations or architecture changes. (§3.4).

## 3.2 Task

We are given a corpus $\mathcal{D} = \{(\boldsymbol{x}_t, \mathbb{Y}_t)\}_{t=1}^{m}$ where $\boldsymbol{x}_t$ is a sequence of tokens and $\mathbb{Y}_t = \{\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k\}$ is a set. For example, in multi-label fine-grained sentiment classification, $\boldsymbol{x}_t$ is a paragraph, and $\mathbb{Y}_t$ is a set of sentiments expressed by the paragraph. We use $\mathrm{y}_i$ to denote an output symbol, $[\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k]$ to denote an ordered sequence of symbols and $\{\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k\}$ to denote a set.

### 3.2.1 Set generation using SEQ2SEQ model

**Task** Given a corpus $\{(\boldsymbol{x}_t, \mathbb{Y}_t)\}_{t=1}^{m}$, the task of conditional set generation is to efficiently estimate $p(\mathbb{Y}_t \mid \boldsymbol{x}_t)$. SEQ2SEQ models factorize $p(\mathbb{Y}_t \mid \boldsymbol{x}_t)$ autoregressively (AR) using the chain rule:

$$\begin{aligned} p(\mathbb{Y}_t \mid \boldsymbol{x}_t) &= p(\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k \mid \boldsymbol{x}_t) \\ &= p(\mathrm{y}_1 \mid \boldsymbol{x}_t) \prod_{j=2}^{k} p(\mathrm{y}_j \mid \boldsymbol{x}_i, \mathrm{y}_1 \ldots \mathrm{y}_{j-1}) \end{aligned}$$

where the order $\mathbb{Y}_t = [\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k]$ factorizes the joint distribution using chain rule. In theory, any of the $k!$ orders can be used to factorize the same joint distribution. In practice, the choice of order is important. For instance, Vinyals et al. [2016] show that output order affects language modeling performance when using LSTM based SEQ2SEQ models for set generation.

Consider an example input-output pair $(\boldsymbol{x}_t, \mathbb{Y}_t = \{\mathrm{y}_1, \mathrm{y}_2\})$. By chain rule, we have the following equivalent factorizations of this sequence: $p(\mathbb{Y}_t \mid \boldsymbol{x}_t) = p(\mathrm{y}_1 \mid \boldsymbol{x})p(\mathrm{y}_2 \mid \boldsymbol{x}, \mathrm{y}_1) = p(\mathrm{y}_2 \mid \boldsymbol{x})p(\mathrm{y}_1 \mid \boldsymbol{x}, \mathrm{y}_2)$. However, order-invariance is only guaranteed with *true* conditional probabilities, whereas the conditional probabilities used to factorize a sequence are *estimated* by a model from a corpus. Further, one of the two factorizations might closely approximate the true distribution, thus being a better choice.

## 3.3 Method

This section expands on two critical components of our system, TSAMPLE. Section 3.3.1 presents TSAMPLE, a novel method to create informative orders over sets tractably. Section 3.3.2 presents our method for jointly modeling cardinality and set output.

### 3.3.1 TSAMPLE: Adding informative orders for set output

SEQ2SEQ formulation requires the output to be in a sequence. Prior work [Vinyals et al., 2016, Rezatofighi et al., 2018, Chen et al., 2021e] has noted that listing the output in orders that have

the highest conditional likelihood given the input is an optimal choice. Unlike these methods, we sidestep exhaustive searching during training using our proposed approach TSAMPLE.

Our core insight is that knowing the optimal order between pairs of symbols in the output drastically reduces the possible number of permutations. We thus impose pairwise order constraints for a subset of labels. Specifically, given an output set $\mathbb{Y}_t = \mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k$, if $\mathrm{y}_i, \mathrm{y}_j$ are independent, they can be added in an arbitrary order. Otherwise, an order constraint is added to the order between $\mathrm{y}_i, \mathrm{y}_j$.

**Learning pairwise constraints** We estimate the dependence between elements $\mathrm{y}_i, \mathrm{y}_j$ using pointwise mutual information: $\texttt{pmi}(\mathrm{y}_i, \mathrm{y}_j) = \log p(\mathrm{y}_i, \mathrm{y}_j)/p(\mathrm{y}_i)p(\mathrm{y}_j)$. Here, $\texttt{pmi}(\mathrm{y}_i, \mathrm{y}_j) > 0$ indicates that the labels $\mathrm{y}_i, \mathrm{y}_j$ co-occur more than would be expected under the conditions of independence [Wettler and Rapp, 1993]. We use $\texttt{pmi}(\mathrm{y}_i, \mathrm{y}_j) > \alpha$ to filter our such pairs of dependent pairs, and perform another check to determine if the order between them should be fixed. For each dependent pair $\mathrm{y}_i, \mathrm{y}_j$, the order is constrained to be $[\mathrm{y}_i, \mathrm{y}_j]$ ($\mathrm{y}_j$ should come after $\mathrm{y}_i$) if $\log p(\mathrm{y}_j \mid \mathrm{y}_i) - \log p(\mathrm{y}_i \mid \mathrm{y}_j) > \beta$, and $[\mathrm{y}_j, \mathrm{y}_i]$ otherwise. Intuitively, $\log p(\mathrm{y}_j \mid \mathrm{y}_i) - \log p(\mathrm{y}_i \mid \mathrm{y}_j) > \beta$ implies that knowledge that a set contains $\mathrm{y}_i$, increases the probability of $\mathrm{y}_j$ being present. Thus, fixing the order to $[\mathrm{y}_i, \mathrm{y}_j]$ will be more efficient for generating a set with $\{\mathrm{y}_i, \mathrm{y}_j\}$.

**Generating samples** To systematically create permutations that satisfy these constraints, we construct a topological graph $G_t$ where each node is a label $\boldsymbol{y}_i \in \mathbb{Y}_t$, and the edges are determined using the $\texttt{pmi}$ and the conditional probabilities as outlined above (Algorithm 1). The required permutations can then be generated as topological traversals $G_t$ (Figure 3.2). We begin the traversal from a different starting node to generate diverse samples. We call this method TSAMPLE. Our method of generating graphs avoids cycles by design (proof in B.4), and thus topological sort remains well-defined. We show that TSAMPLE can be interpreted as a proposal distribution in variational inference framework, which distributes the mass uniformly over informative orders constrained by the graph.

**Do pairwise constraints hold for longer sequences?** While TSAMPLE uses pairwise (and not higher-order) constraints for ordering variables, we note that the pairwise checks remain relevant with extra variables. First, dependence between pair of variables is retained in joint distributions involving more variables ($\mathrm{y}_i \not\perp\!\!\!\perp \mathrm{y}_j \implies \mathrm{y}_i \not\perp\!\!\!\perp \mathrm{y}_j, \boldsymbol{y}_k$) for some $\boldsymbol{y}_k \in \mathbb{Y}$ (Appendix B.1). Further, if $\mathrm{y}_i, \mathrm{y}_j \perp\!\!\!\perp \boldsymbol{y}_k$, then it can be shown that $p(\mathrm{y}_i \mid \mathrm{y}_j) > p(\mathrm{y}_j \mid \mathrm{y}_i) \implies p(\mathrm{y}_i \mid \mathrm{y}_j, \boldsymbol{y}_k) > p(\mathrm{y}_j \mid \mathrm{y}_i, \boldsymbol{y}_k)$ (Appendix B.2). The first property shows that the pairwise dependencies hold in the presence of other set elements. The second property shows that an informative order continues to be informative when additional independent symbols are added. Thus, using pairwise dependencies between the set elements is still effective. Using higher-order dependencies might be suboptimal for practical reasons: higher-order dependencies (or including $\boldsymbol{x}_t$) might not be accurately discovered due to sparsity, and thus cause spurious orders.

Finally, we note that if all the labels are independent, then the order is guaranteed not to matter (Lemma B.3. Thus, our method will only be useful when labels have some degree of dependence.

**Algorithm 1** Generating permutations for $\mathbb{Y}_t$

**Input**: Set $\mathbb{Y}_t$, number of permutations $n$
**Parameter**: $\alpha, \beta$
**Output**: $n$ topological sorts over $G_t(V, E)$

Let $V = \mathbb{Y}_t, E = \emptyset$.
**for all** $\mathrm{y}_i, \mathrm{y}_j \in \mathbb{Y}_t$ **do**
    **if** $pmi(\mathrm{y}_i, \mathrm{y}_j) > \alpha; \lg p(\mathrm{y}_i \mid \mathrm{y}_j) - \lg p(\mathrm{y}_j \mid \mathrm{y}_i) > \beta$ **then**
        $E = E \cup \mathrm{y}_j \rightarrow \mathrm{y}_i$
    **end if**
**end for**
**return** `topo_sort`$(G_t(V, E), n)$

**Complexity analysis** Let $\mathbb{Y}$ be the label space, $(\boldsymbol{x}_t, \mathbb{Y}_t)$ be a particular training example, $N$ be the size of the training set, and $c$ be the maximum number of elements for any set $\mathbb{Y}_t$ in the input. Our method requires three steps: i) iterating over training data to learn conditional probabilities and pmi, and ii) given a $\mathbb{Y}_t$, building the graph $G_t$ (Algorithm 1), and iii) doing topological traversals over $G_t$ to create samples for $(\boldsymbol{x}_t, \mathbb{Y}_t)$.

The time complexity of the first operation is $\mathcal{O}(Nc^2)$: for each element of the training set, the pairwise count for each pair $\mathrm{y}_i, \mathrm{y}_j$ and unigram count for each $\mathrm{y}_i$ is calculated. The pairwise counts can be used for calculating joint probabilities. In principle, we need $\mathcal{O}(|\mathbb{Y}|^2)$ space for storing the joint probabilities. In practice, only a small fraction of the combinations will appear $|\mathbb{Y}|^2$ in the corpus.

Given a set $\mathbb{Y}_t$, the graph $G_t$ is created in $\mathcal{O}(c^2)$ time. Then, generating $k$ samples from $G_t$ requires a topological sort, for $\mathcal{O}(kc)$ (or $\mathcal{O}(c)$ per traversal). For training data of size $N$, the total time complexity is $\mathcal{O}(Nck)$. The entire process of building the joint counts and creating graphs and samples takes less than five minutes for all the datasets on an 80-core Intel Xeon Gold 6230 CPU.

**Interpreting TSAMPLE as a proposal distribution over orders** We show that our method of augmenting permutations to the training data can be interpreted as an instance of variational inference with the order as a latent variable, and TSAMPLE as an instance of a richer proposal distribution.

Let $\pi_j$ be the $j^{th}$ order over $\mathbb{Y}_t$ (out of $|\mathbb{Y}_t|!$ possible orders $\Pi$), and $\pi_j(\mathbb{Y}_t)$ be the sequence of elements in $\mathbb{Y}_t$ arranged with order $\pi_j$. Treating $\pi$ as a latent random variable, the output distribution can then be recovered by marginalizing over $\Pi$: $\log p_\theta(\mathbb{Y}_t \mid \boldsymbol{x}_t) = \log \sum_{\pi_z \in \Pi} p_\theta(\pi_z(\mathbb{Y}_t) \mid \boldsymbol{x}_t)$, $\Pi$: $\log p_\theta(\mathbb{Y}_t \mid \boldsymbol{x}_t) = \log \sum_{\pi_z \in \Pi} p_\theta(\mathbb{Y}_t, \pi_z \mid \boldsymbol{x}_t)$ where $p_\theta$ is the SEQ2SEQ conditional generation model. While summing over $\Pi$ is intractable, standard techniques from the variational inference framework allow us to write a lower bound (ELBO) on the actual likelihood:

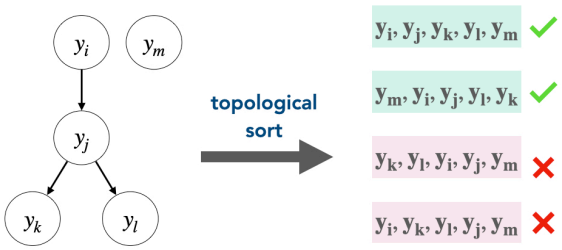



Figure 3.2: Our sampling method TSAMPLE first builds a graph $G_t$ over the set $\mathbb{Y}_t$, and then samples orders from $G_t$ using topological sort (`topo_sort`). The topological sorting rejects samples that do not follow the conditional probability constraints.

$$\log p_\theta(\mathbb{Y}_t \mid \boldsymbol{x}_t) = \log \sum_{\pi_{\boldsymbol{z}} \in \Pi} p_\theta(\pi_z(\mathbb{Y}_t) \mid \boldsymbol{x}_t)$$
$$\geq \underbrace{\mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\frac{\log p_\theta(\pi_{\boldsymbol{z}}(\mathbb{Y}_t) \mid \boldsymbol{x}_t)}{q_\phi(\pi_{\boldsymbol{z}})}]}_{\text{ELBO}} = \mathcal{L}(\theta, \phi)$$

In practice, the optimization procedure draws $k$ samples from the proposal distribution $q$ to optimize a weighted ELBO [Burda et al., 2016, Domke and Sheldon, 2018]. Crucially, $q$ can be fixed (e.g., to uniform distribution over the orders), and in such cases only $\theta$ are learned (Appendix B.6).

TSAMPLE can thus be seen as a particular proposal distribution that assigns all the support to the topological ordering over the label dependence graphs. We experiment with sampling from a uniform distribution over the samples (referred to as RANDOM experiments in our baseline setup). The idea of using an informative proposal distribution over space of structures to do variational inference has also been used in the context of grammar induction [Dyer et al., 2016] and graph generation [Jin et al., 2018, Chen et al., 2021e]. Our formulation is closest in spirit to Chen et al. [2021e]. However, the set of nodes to be ordered is already given in their graph generation setting. In contrast, we infer the order and the set elements jointly from the input.

### 3.3.2 Modeling cardinality

Let $m = |\mathbb{Y}_t|$ be the cardinality of $\mathbb{Y}_t$ (or the number of elements in $\mathbb{Y}_t$). Our goal is to jointly estimate $m$ and $\mathbb{Y}_t$ (i.e., $p(m, \mathbb{Y}_t \mid \boldsymbol{x}_t)$). Additionally, the model must use the cardinality information for generating $\mathbb{Y}_t$. We add the order information at the beginning of the sequence by converting a sample $(\boldsymbol{x}_t, \mathbb{Y}_t)$ to $(\boldsymbol{x}_t, [|\mathbb{Y}_t|, \pi(\mathbb{Y}_t)])$, and then train our SEQ2SEQ model as usual from $\boldsymbol{x} \rightarrow [|\mathbb{Y}_t|, \pi(\mathbb{Y}_t)]$. As SEQ2SEQ models use autoregressive factorization, listing the order information first ensures that the sequence factorizes as $p([|\mathbb{Y}_t|, \pi(\mathbb{Y}_t)] \mid \boldsymbol{x}_t) = p(|\mathbb{Y}_t| \mid \boldsymbol{x}_t)p(\pi(\mathbb{Y}_t) \mid |\mathbb{Y}_t|, \boldsymbol{x}_t)$. Thus, the generation of $\mathbb{Y}_t$ is conditioned on the input and the cardinality (note the $p(\pi(\mathbb{Y}_t) \mid |\mathbb{Y}_t|, \boldsymbol{x}_t)$ term).

**Why should cardinality help?** Unlike models like deep sets [Zhang et al., 2019b], SEQ2SEQ models are not restricted by the number of elements generated. However, adding cardinality information has two potential benefits: i) it can help avoid over-generation [Welleck et al., 2020, Fu et al., 2021], and ii) unlike free-form text output, the distribution of the set output size ($p(|\mathbb{Y}_t| \mid \boldsymbol{x}_t)$) might benefit the model to adhere to the set size constraint.

## 3.4 Experiments

TSAMPLE comprises: i) TSAMPLE, a way to generate informative orders to convert sets to sequences, and ii) CARD: jointly modeling cardinality and the set output. This section answers two questions:

RQ1: **How well does TSAMPLE improve existing models?** Specifically, how well TSAMPLE can take an existing SEQ2SEQ model and improve it just using our data augmentation and joint cardinality prediction, without making any changes to the model architecture. We also measure if these performance improvements carry across diverse datasets, model classes, and inference settings.

RQ2: **Why does our approach improve performance?** We study the contributions of TSAMPLE and joint cardinality prediction (CARD), and analyze where TSAMPLE works or fails.

### 3.4.1 Setup

**Tasks** We consider multi-label classification and keyphrase generation. These tasks represent set generation problems where the label space spans a set of fixed categories (multi-label classification) or free-form phrases (keyphrase generation).

1. **Multi-label classification task**: We have three datasets of varying sizes and label space:
   - Go-Emotions classification (GO-EMO, Demszky et al. [2020]): generate a set of emotions for a paragraph.
   - Open Entity Typing (OPENENT, Choi et al. [2018]): assigning open types (free-form phrases) to the tagged entities in the input text.
   - Reuters-21578 (REUTERS, Lewis [1997]): labeling news article with the set of mentioned economic subjects.
2. **Keyphrase generation (KEYGEN)**: We experiment with a popular keyphrase generation dataset, KP20K [Meng et al., 2017] which involves generating keyphrases for a scientific paper abstract.

Table 3.1 lists the dataset statistics and examples from each dataset are shown in Appendix B.4. We treat all the problems as open-ended generation, and do not use any specialized pre-processing. For all the datasets, we filter out samples with a single label. For each training sample, we create $n$ permutations using TSAMPLE.

**Baselines** We compare with two baselines:

i) **MULTI-LABEL**: As a non-SEQ2SEQ baseline, we train a multi-label classifier that makes independent predictions of the output labels. Encoder-only and encoder-decoder approaches can be

| Task | Avg/min/max labels per sample | Unique labels | Train/test/dev samples per split |
|---|---|---|---|
| GO-EMO | 3.03/3/5 | 28 | 0.6k/0.1k/0.1k |
| OPENENT | 5.4/2/18 | 2519 | 2k/2k/2k |
| REUTERS | 2.52/2/11 | 90 | 0.9k/0.4k/0.3k |
| KEYGEN | 3.87/3/79 | 274k | 156k/2k/2k |

Table 3.1: Datasets used in our experiments.

adapted for MULTI-LABEL, and we experiment with BART (encoder-decoder) and BERT (encoder-only). This baseline represents a standard method for doing multi-label classification (e.g., Demszky et al. [2020]). During inference, top-k logits are returned as the predicted set. We search over $k = [1, 3, 5, 10, 50]$ and use $k$ that performs the best on the dev set. Table 16 in Appendix B.5 shows precision, recall, and $F$ scores at each-k.

ii) **SET SEARCH**: each training sample $(\boldsymbol{x}, \{\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_k\})$ is converted into $k$ training examples $\{(\boldsymbol{x}, \mathrm{y}_i)\}_{i=1}^{k}$. We fine-tune BART-base to generate one training sample for input $\boldsymbol{x}$. During inference, we run beam-search with the maximum set size in the training data (Table 3.1). The unique elements generated by beam search are returned as the set output, a popular approach for one-to-many generation tasks [Hwang et al., 2021].

iii) TSAMPLE can apply to *any* SEQ2SEQ model. We show results with models of various capacity: [iii)] BART-base [Lewis et al., 2020a] (110M), T5 [Raffel et al., 2020c] (11B), and GPT-3 [Brown et al., 2020a] (175B).

**Training** We augment $n = 2$ permutations to the original data using TSAMPLE. For all the results, we use three epochs and the same number of training samples (i.e., input data for the baselines is oversampled). This controls for models trained with augmented data improving only because of factors such as longer training time. All the experiments were repeated for three different random seeds, and we report the averages. We found from our experiments[1] that hyperparameter tuning over $\alpha, \beta$ did not affect the results in any significant way. For all the experiments reported, we use $\alpha = 1$ and $\beta = \log_2(3)$. We use a single GeForce RTX 2080 Ti for all our experiments on bart, and a single TPU for all experiments done with T5-11B. For GPT-3, we use the OpenAI completion engine (davinci) API [OpenAI, 2021]. Additional hyperparameter details in Appendix B.3. We use greedy sampling for all experiments.

### 3.4.2 TSAMPLE improves existing models

Our method helps across a wide range of models (BART, T5, and GPT-3) and tasks.

[1]We conduct a one-tailed proportion of samples test [Johnson et al., 2000] to compare with the strongest baseline, and underscore all results that are significant with $p < 0.0005$. For Algorithm 1, we try $\alpha = \{0.5, 1, 1.5\}$ and $\beta = \{\log_2(2), \log_2(3), \log_2(4)\}$, and use networkx implementation of topological sort [Hagberg et al., 2008a].

| | GO-EMO | OPENENT | REUTERS |
|---|---|---|---|
| SET SEARCH (BART) | 7.4 | 26.3 | 7.5 |
| MULTI-LABEL (BART) | 25.6 | 16.4 | 25.2 |
| MULTI-LABEL (BERT) | 25.7 | 16.2 | 25.5 |
| BART | 23.4 | 44.6 | 15.6 |
| BART + TSAMPLE | **30.0** | ***53.5*** | ***26.7*** |
| T5 | 47.8 | 53.6 | 45.3 |
| T5 + TSAMPLE | **50.9** | ***57.0*** | ***48.5*** |

Table 3.2: TSAMPLE improves SEQ2SEQ models by $\sim$20% relative $F1$- points, on three multilabel classification datasets. BART and T5 are trained on the original datasets with a random order and no cardinality. "+ TSAMPLE" indicates augmented train data using TSAMPLE and cardinality is prepended to the output sequence.

### Multi-label classification

Table 3.2 shows improvements across all datasets and models for the multi-label classification task ($\sim$20% relative gains). For brevity, we list macro $F$ score, and include detailed results including macro/micro precision, recall, $F$ scores in Table 12 (Appendix B.5). We attribute the comparatively lower performance of SET SEARCH baseline to two specific reasons - repeated generation of the same set of terms (e.g., *person, business* for OPENENT) and generating elements not present in the test set (see Section 3.4.3 for a detailed error analysis). We see similar trends with GPT-3 (3.4.2).

### Keyphrase generation

To further motivate the utility of SEQ2SEQ models for set generation tasks, we experiment on KP-20k, which is an extreme multi-label classification dataset [Meng et al., 2017] with label space spanning over 257k unique keyphrases. Due to the large label space, training multi-class classification baselines is not computationally viable. In this dataset, the input text is an abstract from a scientific paper. We use the splits used by Ye et al. [2021]. For a fair comparison with Ye et al. [2021], we use BART-base for this experiment. Table 3.3 shows the results. Similar to datasets with smaller label space, our method improves on vanilla SEQ2SEQ.

We want to emphasize that while specialized models for individual tasks might be possible, we aim to propose a general approach that shows that sampling informative orders can help efficient and general set-generation modeling.

| Ye et al. [2021] | BART | BART + TSAMPLE |
|---|---|---|
| 5.8 | 5.3 | 6.5 |
| 39.2 | 36.3 | 39.1 |

Table 3.3: TSAMPLE improves off-the-shelf BART-base for keyphrase generation task

**Few-shot prompting with GPT-3**

We fine-tune the generation models using augmented data for both BART and T5. However, fine-tuning models at the scale of GPT-3 is prohibitively expensive. Thus such models are typically used in *a few-shot prompting setup*.[2] Our approach is the only feasible candidate for such settings, as it does not involve changing the model or additional post-processing. We apply our approach for tuning prompts for generating sets in few-shot settings. We focus on GO-EMO and OPENENT tasks, as the relatively short input examples allow cost-effective experiments. We randomly create a prompt with $M = 24$ examples from the training set and run inference over the test set for each. For each example in the prompt, we order the set of emotions using our ordering approach TSAMPLE and compare the results with random orderings. Using TSAMPLE to arrange the labels outperforms random ordering for both OPENENT (macro $F$ 34 vs. 39.5 with ours, 15% statistically significant relative improvement), and GO-EMO (macro $F$ 16.5 vs. 14.5, 14% relative improvement). This suggests that ordering helps performance in resource-constrained settings e.g., few-shot prompting.

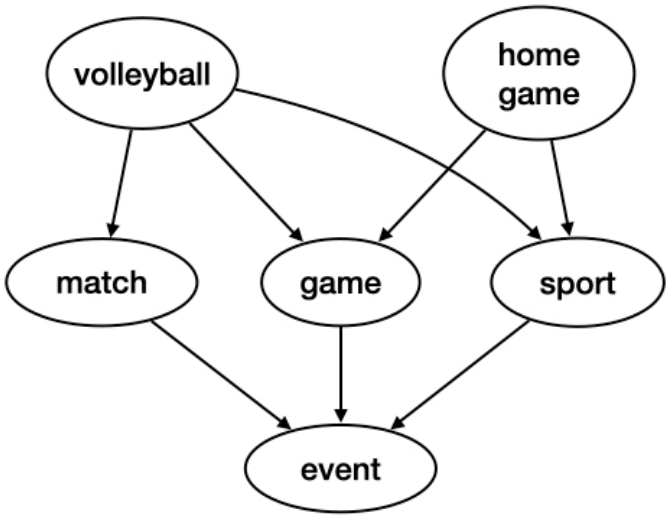



Figure 3.3: Label dependency discovered by TSAMPLE for OPENENT: specific entities (e.g., volleyball) precede generic ones (event). Appendix B.2 has more examples

### 3.4.3 Why does TSAMPLE improve performance?

As mentioned in Section 7.2, our method of generating sets with SEQ2SEQ models consists of two components: i) a strategy for sampling informative orders over label space (TSAMPLE), and ii) jointly generating cardinality of the output (CARD). This section studies the individual contributions of these components in order to answer RQ2.

[2] In a few-shot prompting setup, $M$ ($\sim$10-100) input-output examples are selected as a prompt $p$. A new input $x$ is appended to the prompt $p$, and $p||x$ is the input to GPT3.

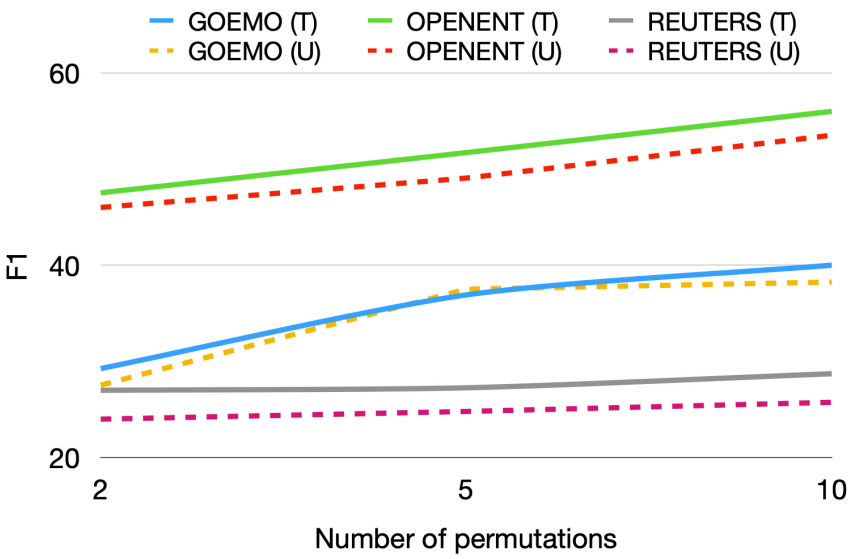



Figure 3.4: TSAMPLE (T) consistently outperforms RANDOM (U) as the number of permutations ($n$) is increased.

### Ablation study

We ablate the two critical components of our system: cardinality (TSAMPLE- CARD) and order (TSAMPLE- TSAMPLE) and investigate the performance for each of these settings using BART for multi-label classification. Table 3.4 presents the results. Both the components individually help, but a larger drop is seen by removing cardinality. We also train using RANDOM orders, instead of TSAMPLE. RANDOM does not improve over SEQ2SEQ consistently (both with and without CARD), showing that merely augmenting with random permutations does not help.

| | GO-EMO | OPENENT | REUTERS |
|---|---|---|---|
| TSAMPLE | **30.0** | **53.5** | **26.7** |
| TSAMPLE- CARD | 23.3 (-22%) | 48.0 (-10%) | 15.8 (-40%) |
| TSAMPLE- TSAMPLE | 26.8 (-11%) | 50.5 (-6%) | 24.3 (-9%) |
| RANDOM | 27.5 (-8%) | 50.4 (-6%) | 24.7 (-7%) |

Table 3.4: Ablations: modeling cardinality (CARD) and sampling informative orders (TSAMPLE) both help, with larger gains from CARD. RANDOM ordering hurts.

### Role of order

**Nature of permutations created by TSAMPLE** TSAMPLE encourages highly co-occurring pairs $(\mathrm{y}_i, \mathrm{y}_j)$ to be in the order $\mathrm{y}_i, \mathrm{y}_j$ if $p(\mathrm{y}_j \mid \mathrm{y}_i) > p(\mathrm{y}_i \mid \mathrm{y}_j)$. In our analysis, this dependency in the datasets shows that the orders exhibit a pattern where *specific* labels appear before the *generic* ones. E.g., in entity typing, the more generic entity *event* is generated after the more specific entities *home game* and *match* (see Figure 3.3).

**Increasing # permutations ($n$) helps:** Fig. 3.4 shows that TSAMPLE and RANDOM improve as $n$ is increased from $n = 2$ to 10; TSAMPLE outperforms RANDOM across $n$.

**Reversing the order hurts performance** In order to check our hypothesis of whether only informative orders helping with set generation, we invert the label dependencies returned by TSAMPLE for all the datasets and train with the same model settings. Across all datasets, we observe that reversing the order leads to an average of 12% drop in $F1$- score. The reversed order not only closes the gap between TSAMPLE and RANDOM, but in many instances, the performance is slightly worse than RANDOM.

### Role of cardinality

**Cardinality is successfully predicted and used** Table 3.4 shows that cardinality is crucial to modeling set output. To study whether the models learn to condition on predicted cardinality, we compute an *agreement* score - defined as the % of times the predicted cardinality matches the number of elements generated by the model. The model effectively predicts the cardinality almost exactly in GO-EMO and REUTERS datasets (avg. 95%). While the exact match agreement is low in OPENENT (35%), the model is within an error of $\pm 1$ in 93% of the cases. These results show that cardinality predicts the end of sequence (EOS) token. The accuracy for predicting the exact cardinality is 61% across datasets, and it increases to 76% within an error of 1 SD.

**Information about cardinality improves multi-label classification** MULTI-LABEL baseline uses different values of k for predicting labels. To test if knowledge of cardinality improves multi-class classification, we experiment with a setting where the true cardinality is available at inference (i.e., $k$ is set to the true value of cardinality). Table 3.5 shows that cardinality improves performance.

| | GO-EMO | OPENENT | REUTERS |
|---|---|---|---|
| MULTI-LABEL | **22.4** | 14.3 | 21.7 |
| MULTI-LABEL-K* | 21.3(-4.9%) | **17.8**(+24.5%) | **25.6**(+18%) |

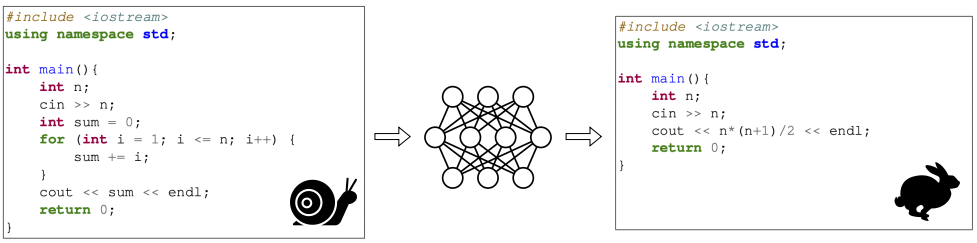
Table 3.5: Cardinality improves multi-label classification.

### Error analysis

We manually compare the outputs generated by the vanilla BART model with BART + TSAMPLE. For the open-entity typing dataset, we randomly sample 100 examples and find that vanilla SEQ2SEQ approach generates sets with an ill-formed element 22% of the time, whereas TSAMPLE completely avoids this. Examples of such ill-formed elements include *personformer*, *businessirm*, *polit*, *foundationirm*, *politplomat*, *eventlete*. This analysis indicates that training the model with an informative order infuses more information about the underlying type-hierarchy, avoiding the ill-formed elements.

## 3.5 Conclusion

We present a novel method for performing conditional set generation using SEQ2SEQ models that leverages both incorporating informative orders and adding cardinality information. Experiments in simulated settings and real-world datasets show that our method is more effective than strong baselines at set generation. TSAMPLE is a computationally efficient and general-purpose plug-in data augmentation algorithm that improves SEQ2SEQ models for set generation in a wide array of settings.




(a) Slower Code.

(b) Faster Code.

Figure 4.1: An example of a program that solves the problem of "*compute the sum of the numbers from 1 to N*". The program in Figure 4.1(a) runs in $\Theta(N)$, whereas the program in Figure 4.1(b) runs in constant time complexity.

# Chapter 4

# Learning to Generate Performance Enhancing Code Edits

## 4.1 Introduction

Despite the impressive progress of optimizing compilers and other tools for performance engineering [Aho et al., 2007], programmers are still largely responsible for high-level performance considerations such as algorithms and API choices. Recent work has demonstrated the promise of deep learning for automating performance optimization [Garg et al., 2022, Mankowitz et al., 2023]. However, these techniques are either narrow or difficult to build on due to the lack of open datasets and lack of reliable performance measurement techniques, which has stymied research in this direction. Recently, pre-trained large language models (LLMs) have demonstrated impressive performance at a wide range of programming tasks [Chen et al., 2021c, Fried et al., 2022b, Xu et al., 2022b, Nijkamp et al., 2022b]. Yet, the effectiveness of large, pre-trained LLMs for program optimization remains an open research question. We study whether such LLMs can be adapted for performance optimization. To this end, we introduce a novel benchmark for performance optimization that addresses the key challenge of replicable performance measurement, and perform an extensive evaluation of a wide range of adaptation techniques based on it.

First, we construct a dataset of **P**erformance-**I**mproving **E**dits (GPT-2). We collect C++

programs written to solve competitive programming problems, where we track a single programmer's submissions as they evolve over time, filtering for sequences of edits that correspond to performance improvements.

Next, a major challenge is the significant variability of measuring performance on real hardware due to server workload and configuration issues. Indeed, we find that benchmarking on real hardware can lead to large, phantom performance "improvements" due only to random chance. To address this challenge, we evaluate performance using the gem5 CPU simulator [Binkert et al., 2011], the gold standard CPU simulator in academia and industry, and models state-of-the-art general-purpose processors. This evaluation strategy is entirely deterministic, ensuring both reliability and reproducibility.

Based on this benchmark, we evaluate a variety of techniques for adapting pre-trained code LLMs for performance optimization. First, we consider baseline prompting approaches, including techniques such as chain-of-thought [Wei et al., 2022d] (CoT). We find that LLMs are limited in the challenging task of code optimization. Without data-driven methods that leverage GPT-2, our strongest baseline CoT only warrants a 1.6× average speedup over 8 submissions vs. the 3.65× human reference. Next we consider a retrieval-based prompting approach where retrieval is used to select examples most similar to the current one [Liu et al., 2022a, Poesia et al., 2022]. Lastly, we consider several finetuning strategies: these include using synthetic data generated via self-play [Haluptzok et al., 2022], where synthetic training examples are generated by an LLM without the need for direct human examples, as well as performance-conditioned generation, where we condition generation on the performance of the generated program.

We find that data-driven methods using GPT-2, like retrieval-based few-shot prompting and fine-tuning, are highly effective at achieving strong optimization abilities in LLMs. When allowing a model to take 8 samples and filtering for correctness and execution time, our fine-tuned performance-conditioned version of CODELLAMA 13B can achieve an average speedup of 5.65× on our test set, and a fine-tuned version of GPT-3.5 augmented with synthetic data via self-play achieves an average speedup of 6.86×, the average human sampled in our test set achieved an average speedup of 3.65×. Aggregating over all human submissions in the test set, these models achieve respective speedups of 9.11× and 9.15× surpassing the best human submission over all available submissions across all programmers we benchmarked.

### 4.1.1 Motivating Example

The example in Figure 4.1 demonstrates the potential of large language models for program optimization. The program in Figure 4.1(a) is a naïve implementation of a program that prints the "*sum of the numbers from 1 to N*", which runs in $\Theta(N)$ – it performs N iterations. However, this problem has a closed-form solution using an arithmetic expression that runs in constant time. When we provided CODEX with the example in Figure 4.1(a) along with a comment: *// Optimize the above program.* we received the program in Figure 4.1(b) as an output. When run with an input of 100,000, the program in Figure 4.1(b) runs over 100x faster than the program in Figure 4.1(a) when compiled with GCC's -O3 optimization level. Without prior knowledge of the formula, it may be non-trivial for an optimizer to propose or even prove the equivalence between the two programs. In contrast, LLMs that were trained on vast amounts of code may have implicitly learned to write efficient code. In this work, we explore the potential of large

language models to improve programs in a similar fashion beyond such contrived examples. We also investigate how to improve the optimization ability of these large language models.

## 4.2 Performance Improving Edits (PIE) Dataset

We construct a dataset targeted at adapting code LLMs to performance optimization, focusing on optimizing program execution time. Our dataset is constructed based on performance-improving edits (PIE) made by human programmers in a range of competitive programming tasks from CodeNet [Puri et al., 2021a]. We exclusively focus on C++ programs since it is a performance-oriented language compatible with the gem5 simulator. Given a problem, programmers typically write an initial solution and iteratively improve it. Let $\mathbb{Y}^u = [y_1^u, y_2^u, ...]$ be a chronologically sorted series of programs, written by user $u$ for problem $x$. From $\mathbb{Y}^u$, we remove programs that were not accepted by the automated system, eliminating incorrect programs (fail one or more unit tests) or take more than the allowed time to run, resulting in a *trajectory* of programs $\mathbb{Y}^* = [y_1^*, y_2^*, \ldots, y_n^*]$.

For each trajectory $\mathbb{Y}^*$, we construct pairs $\mathbb{P} = (y_1, y_2), (y_1, y_3), (y_2, y_3) \ldots$, and keep only pairs for which $\frac{(\texttt{time}(y_i) - \texttt{time}(y_{>i}))}{\texttt{time}(y_i)} > 10\%$ where $\texttt{time}\,(y)$ is the measured latency of program $y$ (i.e., the relative time improvement is more than 10%). The CodeNet dataset includes CPU time, but we found the information to be inconsistent (see Appendix D.8). Thus, we relabel the execution time using gem5 as described below; to create these annotated runtimes, we performed over 42.8 million simulations in our gem5 environment.

We split the resulting dataset of pairs $\mathbb{P}$ into train/validation/test sets, ensuring that any particular competitive programming problem only appears in one of them. We obtain a training set of 77,967 pairs from 1,474 problems, a validation set of 2,544 pairs from 77 problems, and a test set of 978 pairs from 41 problems. For each pair in the test set, we also record the fastest human submission execution time for that problem; in Section 4.3.3, we include this running time as a comparison point.

**Test cases.** Our goal is to improve performance while ensuring correctness. We evaluate correctness through unit tests; we reject the program if a single test fails. CodeNet includes an average of 4 test cases per problem. To improve coverage, we include additional test cases from AlphaCode [Li et al., 2021b] generated with a fine-tuned LLM. A small set of test cases would lead to substantial timeouts above 2 minutes in gem5; after excluding them, we obtain a median of 82.5 test cases per problem in our training set, 75 test cases per problem in our validation set, and 104 test cases per problem for our test set. See Appendix D.4 for additional details.

**Performance measurement using gem5.** Benchmarking program performance is notoriously difficult. For instance, code instrumentation introduces overhead, and there is substantial variance across executions due to numerous factors, including server load and idiosyncrasies introduced by the operating system. If benchmarking is not performed carefully, it is easy to mistakenly over-report program optimization results. With enough samples and variance, benchmarking the same exact program can easily lead us to report significant optimizations.

To illustrate the challenges, consider HYPERFINE Peter [2023], a Rust library designed to precisely benchmark binaries. We benchmarked 500 programs "pairs" where the "slow" and "fast" programs are identical. Ideally, we should have $\frac{\text{source time}}{\text{target time}} = 1$ (i.e., the two programs have identical

performance). However, we observed a mean speedup of 1.12×, with a standard deviation of 0.36, and the top 5% of pairs exhibited a speedup of 1.91×. These results underscore the significant challenges in performance measurement.

To address this challenge, we measure program performance using the gem5 [Binkert et al., 2011] full system detailed microarchitectural simulator of state-of-the-art processors. Executing deterministic programs in gem5 provides fully deterministic performance results, enabling reliable isolation of the impact of performance-improving edits and reproducibility. We use the `Verbatim` configuration of the Intel Skylake architecture from gem5.[1]. An advantage of this approach is that our framework can be applied to other platforms like ARM or RISC-V without having access to hardware for those platforms.

## 4.3 Learning to Improve Code Performance

### 4.3.1 Few-Shot Prompting

**Instruction-prompting.** We use prompts instructing the LLM to improve the performance of the given program, an approach commonly referred to as instruction prompting [Mishra et al., 2022b, Gupta et al., 2022, Longpre et al., 2023]; details on the prompt are in Figure 24 in Appendix D.10.

**Few-shot prompting.** Next, we use few-shot prompting [Brown et al., 2020b]. In particular, we create a prompt with the format "slow$_1$ → fast$_1$ —— slow$_2$ → fast$_2$ —— . . . ". A slow test set program is appended to this prompt during inference and supplied to the model. We create the prompts by randomly sampling two (fast, slow) pairs from the training set. Examples of prompts are shown in Figure 25 in Appendix D.10.

**Chain-of-thought prompting.** Inspired by Chain-of-Thought (CoT) prompting [Wei et al., 2022d], we also designed prompts that ask the LLM to *think about* how to optimize the program before actually producing the optimized program. This strategy is used in conjunction with few-shot prompting. Examples of prompts are shown in Figure 26 in Appendix D.10.

**Dynamic retrieval-based few-shot prompting.** Recent work has demonstrated that retrieval-based mechanisms can improve language models for various tasks requiring factual or procedural knowledge [Liu et al., 2022a, Poesia et al., 2022, Odena and Sutton, 2020, Madaan et al., 2022a, Shrivastava et al., 2023]. Program optimization is a non-trivial task requiring knowledge of algorithms, data structures, and programming grounded in performance; thus, retrieving highly relevant examples may improve an LLM's optimization ability. For example, a solution optimized for a knapsack problem in dynamic programming could inform strategies for the coin change problem. Through dynamic retrieval-based prompts, we aim to match tasks with analogous structures or challenges, allowing models to better harness the patterns in PIE. We use the CodeBertScore models trained for C++ [Zhou et al., 2023b] to embed both the program to be optimized and the programs in PIE. We use FAISS [Johnson et al., 2019a] to retrieve $K$ closest programs from the training set; and to construct a "slow$_1$ → fast$_1$ —— ..." style prompt on the fly. Examples of prompts are shown in Figure 27 in Appendix D.10.

[1] https://github.com/darchr/gem5-skylake-config

### 4.3.2 Finetuning

We also consider fine-tuning to improve pretrained code LLMs using our PIE dataset. In addition to standard fine-tuning on the entire dataset, we describe additional strategies we used.

**Dataset imbalance.** While we have tens of thousands of slow-fast pairs in the PIE training dataset, these submissions target just 1,474 problems, which may limit the learned model's ability to generalize to new programs. Furthermore, submissions are not uniformly distributed across problems. To address this imbalance, we additionally introduce a subset of 4,085 "high-quality" slow-fast pairs—in particular, we take examples with the highest speedup and disallow more than 4 submissions per problem, for an average of 2.77 submissions per problem. Given the high costs of training models through the OpenAI API, we also use this dataset as a base for fine-tuning experiments with GPT-3.

```
This is a slow program we want
↪  to optimize to score
↪  {score_tag}/10.

### Program:
{src_code}

### Optimized Version with score
↪  {score_tag}/10:
```

```
This is a slow program we want
↪  to optimize to score 10/10.

### Program:
{src_code}
```

Figure 4.2: Training (left) and inference (right) prompts for Goal-Conditioned optimization with GPT-2.

**Performance-conditioned generation.** Programs can typically be written in many ways with different performance profiles. Consequently, when training a model to predict performance-improving edits with a large dataset like PIE, it is trained on a mix of large and small improvements, without any information on which improvements are more desirable than others. Inspired by recent prompting strategies [Zhang et al., 2023] and offline-rl [Chen et al., 2021a], we introduce performance tags during training by associating each "fast" program with a tag indicating the optimal achievable performance across all solutions in the dataset. Specifically, the tag indicates how close that program is to peak performance on a binned-scale $\{1, 2, \ldots, 10\}$. We instantiate our tags by categorizing the top 10% of optimized solutions in the dataset for a given task as "10/10", the next 10% as "9/10", and so on. These tags enable the model to discern the relationship between specific problem attributes and their corresponding high-performance solutions (Figure 4.2, left). During inference, we prompt the model with a test input and a maximal score tag "10/10", directing it to generate the most optimal solution (Figure 4.2, right).

**Synthetic data.** Given the high cost of obtaining human-written programs, we also augment our dataset with synthetic examples through a multi-stage process. First, we prompt OpenAI's GPT-3 with examples from the PIE dataset, instructing it to produce new programs that produce different outputs given the same inputs. After filtering out programs producing outputs identical to those in PIE and tracking semantic duplicates among those generated, we obtain 3,314 unique

Table 4.1: **Baselines:** Results for baseline prompting strategies and models for Best@1 and Best@8.

| Method | Model | Best@1 %Opt | Best@1 Speedup | Best@1 %Correct | Best@8 %Opt | Best@8 Speedup | Best@8 %Correct |
|---|---|---|---|---|---|---|---|
| Instruction-Only | CODELLAMA 7B | 0.92% | 1.01× | 23.52% | 5.21% | 1.06× | 68.30% |
| Instruction-Only | CODELLAMA 13B | 0.41% | 1.00× | 10.02% | 2.45% | 1.03× | 40.49% |
| Instruction-Only | CODELLAMA 34B | 2.86% | 1.05× | 44.27% | 18.92% | 1.26× | 84.97% |
| Instruction-Only | GPT-3.5 | 16.26% | 1.20× | 80.67% | 39.16% | 1.54× | 98.77% |
| Instruction-Only | GPT-4 | 8.49% | 1.15× | **93.25%** | 21.17% | 1.31× | 98.77% |
| Few-Shot | CODELLAMA 7B | 2.15% | 1.02× | 43.46% | 9.51% | 1.15× | 85.07% |
| Few-Shot | CODELLAMA 13B | 2.25% | 1.02× | 40.29% | 13.70% | 1.21× | 83.03% |
| Few-Shot | CODELLAMA 34B | 2.66% | 1.02× | 43.97% | 13.70% | 1.16× | 82.62% |
| Few-Shot | GPT-3.5 | 11.45% | 1.13× | 80.98% | 29.04% | 1.38× | 95.91% |
| Few-Shot | GPT-4 | 18.92% | 1.25× | 82.82% | 36.40% | 1.44× | **98.98%** |
| COT | CODELLAMA 7B | 0.82% | 1.01× | 27.40% | 7.46% | 1.13× | 73.31% |
| COT | CODELLAMA 13B | 2.25% | 1.04× | 32.92% | 11.15% | 1.20× | 79.24% |
| COT | CODELLAMA 34B | 3.99% | 1.08× | 30.27% | 19.63% | 1.30× | 78.73% |
| COT | GPT-3.5 | 21.37% | 1.25× | 65.95% | **43.05%** | **1.60**× | 91.72% |
| COT | GPT-4 | **26.99%** | **1.32**× | 63.09% | 42.74% | 1.58× | 84.87% |

synthetic programs and many thousand more duplicates. Next, we generate an optimized version for each synthetic "slow" program using a GPT-3 model that has been fine-tuned on the original PIE dataset. Finally, we retain pairs where the optimized program is at least 5× faster and limit semantic duplicates to three, resulting in 1,485 optimized synthetic examples. This methodology aligns with self-play and self-instruct approachs in neural program synthesis [Haluptzok et al., 2022, Roziere et al., 2023]. We provide additional details on the generation process in Appendix D.5.

### 4.3.3 Results for Few-Shot Prompting

**Baseline few-shot prompting.** Table 4.1 (top) shows results on standard few-shot prompting techniques (Section 4.3.1, prompts are shown in appendix D.10). We find that generic few-shot prompts often yield similar results compared to simple instruction-prompting. For instance, when prompted with instructions alone, both GPT-3.5 and CODELLAMA 34B demonstrated superior %OPT and SPEEDUP metrics. This observation aligns with the findings of Zhao et al. [2021], which highlighted that few-shot examples can sometimes bias the model and lead to an incorrect understanding of the task. In the context of our study, the consistent use of the same fixed prompt might constrain the model to only apply optimization techniques present in the prompt, thereby resulting in sub-optimal performance. Finally, in line with the findings of Wei et al. [2022a] that identified COT prompting as an emergent capability, we observe improvements with this approach over both instruction-tuned and fixed prompt setups, but notably only for the larger CODELLAMA (13B and 34B) and GPT-3.5 models. For CoT prompting; we note that GPT-4 outperforms GPT-3 Best@1 and under-performs GPT-3 Best@8: this may demonstrate a lack of output diversity from GPT-4 despite using the same sampling hyper-parameters.

**Retrieval-based few-shot prompting.** Table 4.1 (bottom) shows results using our dynamic retrieval-based few-shot prompting strategy, with the optimal setting at $K = 2$ retrieved prompts. Extended results for $K \in \{1, 2, 4\}$ are detailed in Appendix D.6. The results show that dynamic few-shot prompting outperforms all the baseline variants, showing that GPT-2 effectively adapts LLMs for program optimization in few-shot settings. We note that increased speedup may, however, come with some cost of correctness.

Table 4.2: **Dynamic retrieval-based few-shot prompting:** Results for dynamic retrieval-based few-shot prompting across models for Best@1 and Best@8.

| | | Best@1 | | | Best@8 | | |
|---|---|---|---|---|---|---|---|
| **Method** | **Model** | %Opt | Speedup | %Correct | %Opt | Speedup | %Correct |
| Dynamic Retrieval, K=2 | CODELLAMA 7B | 4.40% | 1.13× | 20.55% | 16.87% | 1.51× | 55.32% |
| Dynamic Retrieval, K=2 | CODELLAMA 13B | 9.10% | 1.35× | 28.73% | 28.02% | 1.97× | 64.72% |
| Dynamic Retrieval, K=2 | CODELLAMA 34B | 10.22% | 1.27× | 25.87% | 34.25% | 2.28× | 63.19% |
| Dynamic Retrieval, K=2 | GPT-3 | 26.18% | 1.58× | 80.37% | 48.06% | 2.14× | **97.85%** |
| Dynamic Retrieval, K=2 | GPT-4 | **50.00%** | **2.61**× | **80.57%** | **74.74%** | **3.95**× | **97.85%** |

### 4.3.4 Results for Finetuning

**Fine-tuning with GPT-2 substantially improves all models.** We fine-tune CODELLAMA and GPT-3 models on our GPT-2 dataset. Due to the cost of fine-tuning and sampling models through the OpenAI API, we were only able to train GPT-3 on the smaller, high-quality dataset (HQ) in Section 4.3.2. The top of Table 4.3 shows results for traditional fine-tuning on all models. We see substantially stronger results when fine-tuning on the smaller, high-quality dataset. These results reflect the observation that to adapt LLMs, a small set of high-quality examples can elicit strong performance [Zhou et al., 2023a, Chen et al., 2023].

**Performance-conditioned training outperforms fine-tuning.** Table 4.3 shows results for performance-conditioned (PERF-COND) generation (Section 4.3.2). Both fine-tuned CODELLAMA models (7B and 13B) show significant improvements in %OPT and SPEEDUP. These gains highlight how the performance improvement information (Figure 4.2) can enable models to distinguish optimal and sub-optimal solutions, leading to more effective optimizations.

**Synthetic data from self-play marginally improves generalization.** Next, we fine-tuned both CODELLAMA and GPT-3 using our GPT-2 dataset augmented with our synthetic examples. We show results at the bottom of Table 4.3. For CODELLAMA and GPT-3, compared to using no synthetic data, the additional data improves both %OPT and often SPEEDUP, particularly with BEST@1. We believe the small set of synthetic examples helped generalize the fine-tuned model, as evidenced by the higher %OPT. [2] We note that the difference saturates with more samples (See Appendix D.2 for details).

[2]For GPT-3, to be sure the increases came from the type of data and not the quantity of data, we performed an ablation by fine-tuning on the top 5,793 examples from GPT-2 with a maximum of 8 duplicates (instead of the 5,570 pairs that included synthetic programs), and we saw BEST@1 performance degrade %OPT to 36.66% and SPEEDUP to 2.67×, and BEST@8 performance degrade %OPT to 83.63% and SPEEDUP to 6.03×.

Table 4.3: **Fine-Tuning:** Results for various models and dataset configurations.

| Dataset | Model | Best@1 | | | Best@8 | | |
|---|---|---|---|---|---|---|---|
| | | %Opt | Speedup | %Correct | %Opt | Speedup | %Correct |
| All | CODELLAMA 7B | 9.20% | 1.31× | 55.21% | 35.58% | 2.21× | 74.03% |
| All | CODELLAMA 13B | 12.78% | 1.52× | 55.42% | 43.76% | 2.71× | 75.46% |
| HQ | CODELLAMA 7B | 10.33% | 1.40× | **76.38%** | 45.30% | 3.14× | 87.63% |
| HQ | CODELLAMA 13B | 11.55% | 1.43× | 70.55% | 47.75% | 3.43× | 85.07% |
| HQ | GPT-3 | 38.55% | 2.70× | 59.10% | 86.71% | 6.74× | **95.40%** |
| All w/Perf-Cond | CODELLAMA 7B | 25.15% | 2.45× | 34.76% | 56.95% | 4.86× | 63.91% |
| All w/Perf-Cond | CODELLAMA 13B | 32.00% | **2.95**× | 38.55% | 66.56% | 5.65× | 70.96% |
| HQ + Self-Play | CODELLAMA 7B | 15.34% | 1.59× | 75.77% | 46.22% | 3.32× | 87.42% |
| HQ + Self-Play | CODELLAMA 13B | 14.31% | 1.61× | 76.28% | 49.69% | 3.51× | 86.20% |
| HQ + Self-Play | GPT-3 | **45.50%** | **3.02**× | 61.55% | **87.63%** | **6.86**× | 95.09% |

### 4.3.5 Discussion and Key Takeaways

**CODELLAMA VS. GPT-3-175B.** Our results demonstrate that openly available models such as CODELLAMA can be competitive with GPT-3. For prompting, CODELLAMA 34B with dynamic retrieval (34.25% %OPT, 2.28× SPEEDUP for BEST@8) roughly matched the performance of GPT-3 with dynamic retrieval (48.06% %OPT, 2.14× SPEEDUP for BEST@8). With fine-tuning, CODELLAMA 13B with performance-conditioned generation (66.56% %OPT, 5.65× SPEEDUP for BEST@8) approached the performance of GPT-3 with synthetic data (87.63% %OPT, 6.86× SPEEDUP for BEST@8); indeed, we may expect that fine-tuning CODELLAMA 34B using the same strategy would further bridge this gap. These results demonstrate that with the right adaptation strategies, open models can be competitive with private ones.

**Prompting vs. fine-tuning.** Our results demonstrate that while prompting can be an effective way to adapt models (with retrieval), fine-tuning significantly outperforms prompting for models of the same size.

**Effectiveness of retrieval-based few-shot learning.** Our results show that dynamic retrieval provides enormous gains over all other prompting approaches; for instance, it improved the performance of CODELLAMA 34B from 19.63 %OPT, 1.30× SPEEDUP to 34.25% %OPT, 2.28× SPEEDUP for BEST@8.

**Effectiveness of performance-conditioned generation.** We find that performance-conditioned generation is incredibly effective for achieving good performance; in particular, it improved the performance of CODELLAMA 13B from 47.75% %OPT, 3.43× SPEEDUP to 66.56% %OPT, 5.65× SPEEDUP for BEST@8.

**Ineffectiveness of LoRA.** We also experimented with low-rank adaptors (LoRA) [Hu et al., 2021], but they performed significantly worse than end-to-end; see Appendix D.9 for results.

### 4.3.6 Analysis of Generated Code Edits

Next, we study the kinds of edits LLMs make that lead to our performance gains, focusing on our best-performing model, GPT-3 fine-tuned with synthetic data. We manually analyze a randomly sampled set of 120 (source, optimized) program pairs to understand the algorithmic and structural changes responsible for the performance gains. We find that the transformations can be broadly categorized into four kinds: *Algorithmic changes*, *Input/Output operations* (IO), *Data Structure modifications*, and *Miscellaneous adjustments*. *Algorithmic changes* (complex modifications, such as changing recursive methods to dynamic programming, and unexpected ones, such as omitting Binary Indexed Trees for simpler constructs) are most common, comprising ˜34.15% of changes; *Input/Output operations* (e.g., changing 'cin/cout' to 'scanf/printf', efficiently reading strings) comprised ˜26.02%; *Data Structures* (e.g., switching from vectors to arrays) comprised ˜21.14%, and *Miscellaneous* (e.g., code cleanups and constant optimizations) comprised ˜18.70%. Please see Appendix D for details and Appendix D.1 for examples of optimizations made by our model.

## 4.4 Appendix

# Part II

# Structure-Assisted Modeling

In the previous chapter, we explored techniques for infusing structure into data before fine-tuning, taking advantage of inherent structure and domain knowledge to improve model performance. However, there are cases where these techniques may not be sufficient, particularly when we want the model to exhibit specific behaviors or leverage complex structures present in the data. In such scenarios, it is beneficial to employ specialized models or setups that can directly integrate these structures.

In this chapter, we discuss two such cases where specialized models and setups play a crucial role in effectively incorporating structure and domain knowledge:

1. A tag and generate pipeline for politeness and style transfer, which utilizes stylistic attributes to improve content preservation and style transfer accuracy while preserving the meaning of sentences.
2. A hierarchical mixture of experts model for structured situational reasoning using graphs, named CURIOUS, which achieves state-of-the-art performance on three different defeasible reasoning datasets by explicitly modeling problem scenarios before answering queries.

# Chapter 5

# Politeness Transfer: A Tag and Generate Approach

Politeness plays a crucial role in social interaction, and is closely tied with power dynamics, social distance between the participants of a conversation, and gender Brown et al. [1987], Danescu-Niculescu-Mizil et al. [2013]. It is also imperative to use the appropriate level of politeness for smooth communication in conversations Coppock [2005], organizational settings like emails Peterson et al. [2011], memos, official documents, and many other settings. Notably, politeness has also been identified as an interpersonal style which can be decoupled from content Kang and Hovy [2019]. Motivated by its central importance, in this paper we study the task of converting non-polite sentences to polite sentences while preserving the meaning.

Prior work on text style transfer Shen et al. [2017], Li et al. [2018], Prabhumoye et al. [2018], Rao and Tetreault [2018], Xu et al. [2012], Jhamtani et al. [2017] has not focused on politeness as a style transfer task, and we argue that defining it is cumbersome. While native speakers of a language and cohabitants of a region have a good working understanding of the phenomenon of politeness for everyday conversation, pinning it down as a definition is non-trivial Meier [1995]. There are primarily two reasons for this complexity. First, as noted by Brown et al. [1987], the phenomenon of politeness is rich and multifaceted. Second, politeness of a sentence depends on the culture, language, and social structure of both the speaker and the addressed person. For instance, while using "please" in requests made to the closest friends is common amongst the native speakers of North American English, such an act would be considered awkward, if not rude, in the Arab culture Kádár and Mills [2011].

We circumscribe the scope of politeness for the purpose of this study as follows: First, we adopt the data driven definition of politeness proposed by Danescu-Niculescu-Mizil et al. [2013]. Second, we base our experiments on a dataset derived from the Enron corpus Klimt and Yang [2004] which consists of email exchanges in an American corporation. Thus, we restrict our attention to the notion of politeness as widely accepted by the speakers of North American English in a formal setting.

Even after framing politeness transfer as a task, there are additional challenges involved that differentiate politeness from other styles. Consider a common directive in formal communication, "send me the data". While the sentence is not impolite, a rephrasing "could you please send me the data" would largely be accepted as a more polite way of phrasing the same statement

[Danescu-Niculescu-Mizil et al., 2013]. This example brings out a distinct characteristic of politeness. It is easy to pinpoint the signals for *politeness*. However, cues that signal the *absence* of politeness, like direct questions, statements and factuality Danescu-Niculescu-Mizil et al. [2013], do not explicitly appear in a sentence, and are thus hard to objectify. Further, the other extreme of politeness, impolite sentences, are typically riddled with curse words and insulting phrases. While interesting, such cases can typically be neutralized using lexicons. For our study, we focus on the task of transferring the non-polite sentences to polite sentences, where we simply define non-politeness to be the absence of both politeness and impoliteness. Note that this is in stark contrast with the standard style transfer tasks, which involve transferring a sentence from a well-defined style polarity to the other (like positive to negative sentiment).

We propose a *tag* and *generate* pipeline to overcome these challenges. The *tagger* identifies words or phrases potentially indicating non-politeness (e.g., "I need this file *right now*") and replaces them with a tag token. Additionally, the tagger adds tag tokens in positions where phrases characteristic of the target style can be inserted (e.g., "*[tag]* send me the data", where [tag] can be replaced with a phrase indicating a request). The *generator* takes as input the output of the tagger and generates a sentence in the target style. For example, the generator could transform the tagged sentence into "*Could you please* send me the data?". Additionally, unlike previous systems, our system's intermediate outputs are fully realized, making the pipeline interpretable. Finally, if the input sentence is already in the target style, our model won't add any stylistic markers and thus would allow the input to flow as is.

We evaluate our model on politeness transfer as well as 5 additional tasks described in prior work Shen et al. [2017], Prabhumoye et al. [2018], Li et al. [2018] on content preservation, fluency and style transfer accuracy. Both automatic and human evaluations show that our model beats the state-of-the-art methods in content preservation, while either matching or improving the transfer accuracy across six different style transfer tasks(§5.4). The results show that our technique is effective across a broad spectrum of style transfer tasks.

Our methodology is inspired by Li et al. [2018] and improves upon several of its limitations as described in (§5.1).

Our main contribution is the design of politeness transfer task. To this end, we provide a large dataset of nearly 1.39 million sentences labeled for politeness (`https://github.com/tag-and-generate/politeness-dataset`). Additionally, we hand curate a test set of 800 samples (from Enron emails) which are annotated as requests. To the best of our knowledge, we are the first to undertake politeness as a style transfer task. In the process, we highlight an important class of problems wherein the transfer involves going from a neutral style to the target style. Finally, we design a "tag and generate" pipeline that is particularly well suited for tasks like politeness, while being general enough to match or beat the performance of the existing systems on popular style transfer tasks.

## 5.1 Related Work

Politeness and its close relation with power dynamics and social interactions has been well documented Brown et al. [1987]. Recent work Danescu-Niculescu-Mizil et al. [2013] in computational linguistics has provided a corpus of *requests* annotated for politeness curated from Wikipedia and

StackExchange. Niu and Bansal [2018] uses this corpus to generate polite dialogues. Their work focuses on contextual dialogue response generation as opposed to content preserving style transfer, while the latter is the central theme of our work. Prior work on Enron corpus Yeh and Harnly [2006] has been mostly from a socio-linguistic perspective to observe social power dynamics Bramsen et al. [2011], McCallum et al. [2007], formality Peterson et al. [2011] and politeness Prabhakaran et al. [2014]. We build upon this body of work by using this corpus as a source for the style transfer task.

Prior work on style transfer has largely focused on tasks of sentiment modification Hu et al. [2017], Shen et al. [2017], Li et al. [2018], caption transfer Li et al. [2018], persona transfer Chandu et al. [2019], Zhang et al. [2018], gender and political slant transfer Reddy and Knight [2016], Prabhumoye et al. [2018], and formality transfer Rao and Tetreault [2018], Xu et al. [2019]. Note that formality and politeness are loosely connected but independent styles Kang and Hovy [2019]. We focus our efforts on carving out a task for politeness transfer and creating a dataset for such a task.

Current style transfer techniques Shen et al. [2017], Hu et al. [2017], Fu et al. [2018], Yang et al. [2018b], John et al. [2019] try to disentangle source style from content and then combine the content with the target style to generate the sentence in the target style. Compared to prior work, "Delete, Retrieve and Generate" Li et al. [2018] (referred to as DRG henceforth) and its extension Sudhakar et al. [2019] are effective methods to generate outputs in the target style while having a relatively high rate of source content preservation. However, DRG has several limitations: (1) the delete module often marks content words as stylistic markers and deletes them, (2) the retrieve step relies on the presence of similar content in both the source and target styles, (3) the retrieve step is time consuming for large datasets, (4) the pipeline makes the assumption that style can be transferred by deleting stylistic markers and replacing them with target style phrases, (5) the method relies on a fixed corpus of style attribute markers, and is thus limited in its ability to generalize to unseen data during test time. Our methodology differs from these works as it does not require the retrieve stage and makes no assumptions on the existence of similar content phrases in both the styles. This also makes our pipeline faster in addition to being robust to noise.

Wu et al. [2019] treats style transfer as a conditional language modelling task. It focuses only on sentiment modification, treating it as a cloze form task of filling in the appropriate words in the target sentiment. In contrast, we are capable of generating the entire sentence in the target style. Further, our work is more generalizable and we show results on five other style transfer tasks.

## 5.2 Tasks and Datasets

### 5.2.1 Politeness Transfer Task

For the politeness transfer task, we focus on sentences in which the speaker communicates a requirement that the listener needs to fulfill. Common examples include imperatives "*Let's stay in touch*" and questions that express a proposal "*Can you call me when you get back?*". Following Jurafsky et al. [1997], we use the umbrella term "action-directives" for such sentences. The goal of this task is to convert action-directives to polite requests. While there can be more than one way of making a sentence polite, for the above examples, adding gratitude ("*Thanks* and let's stay

in touch") or counterfactuals ("*Could* you please call me when you get back?") would make them polite Danescu-Niculescu-Mizil et al. [2013].

**Data Preparation** The Enron corpus Klimt and Yang [2004] consists of a large set of email conversations exchanged by the employees of the Enron corporation. Emails serve as a medium for exchange of requests, serving as an ideal application for politeness transfer. We begin by pre-processing the raw Enron corpus following Shetty and Adibi [2004]. The first set of pre-processing[1] steps and de-duplication yielded a corpus of roughly 2.5 million sentences. Further pruning[2] led to a cleaned corpus of over 1.39 million sentences. Finally, we use a politeness classifier Niu and Bansal [2018] to assign politeness scores to these sentences and filter them into ten buckets based on the score ($P_0$-$P_9$; Fig. 5.1). All the buckets are further divided into train, test, and dev splits (in a 80:10:10 ratio).

For our experiments, we assumed all the sentences with a politeness score of over 90% by the classifier to be polite, also referred as the $P_9$ bucket (marked in green in Fig. 5.1). We use the train-split of the $P_9$ bucket of over 270K polite sentences as the training data for the politeness transfer task. Since the goal of the task is making action directives more polite, we manually curate a test set comprising of such sentences from test splits across the buckets. We first train a classifier on the switchboard corpus Jurafsky et al. [1997] to get dialog state tags and filter sentences that have been labeled as either action-directive or quotation.[3] Further, we use human annotators to manually select the test sentences. The annotators had a Fleiss's Kappa score ($\kappa$) of 0.77[4] and curated a final test set of 800 sentences.

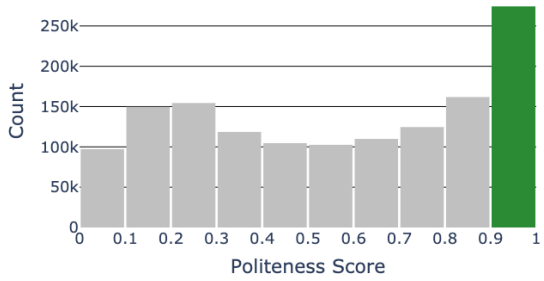



Figure 5.1: Distribution of Politeness Scores for the Enron Corpus

In Fig. 5.2, we examine the two extreme buckets with politeness scores of $< 10\%$ ($P_0$ bucket) and $> 90\%$ ($P_9$ bucket) from our corpus by plotting 10 of the top 30 words occurring in each bucket. We clearly notice that words in the $P_9$ bucket are closely linked to polite style, while words in the $P_0$ bucket are mostly content words. This substantiates our claim that the task of politeness transfer is fundamentally different from other attribute transfer tasks like sentiment where both the polarities are clearly defined.

[1]Pre-processing also involved steps for tokenization (done using spacy Honnibal and Montani [2017]) and conversion to lower case.

[2]We prune the corpus by removing the sentences that 1) were less than 3 words long, 2) had more than 80% numerical tokens, 3) contained email addresses, or 4) had repeated occurrences of spurious characters.

[3]We used AWD-LSTM based classifier for classification of action-directive.

[4]The score was calculated for 3 annotators on a sample set of 50 sentences.

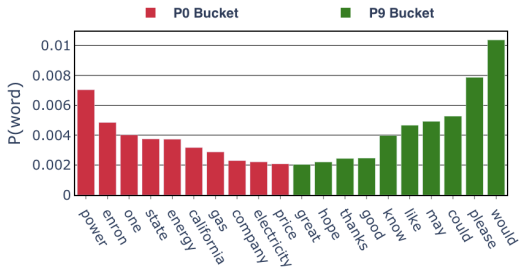



Figure 5.2: Probability of occurrence for 10 of the most common 30 words in the $P_0$ and $P_9$ data buckets


Figure 5.3: Our proposed approach: *tag* and *generate*. The tagger infers the interpretable style free sentence $z(\mathbf{x}_i)$ for an input $\mathbf{x}_i^{(1)}$ in source style $\mathcal{S}_1$. The generator transforms $\mathbf{x}_i^{(1)}$ into $\hat{\mathbf{x}}_i^{(2)}$ which is in target style $\mathcal{S}_2$.

### 5.2.2 Other Tasks

The **Captions** dataset Gan et al. [2017] has image captions labeled as being factual, romantic or humorous. We use this dataset to perform transfer between these styles. This task parallels the task of politeness transfer because much like in the case of politeness transfer, the captions task also involves going from a style neutral (factual) to a style rich (humorous or romantic) parlance.

For sentiment transfer, we use the **Yelp** restaurant review dataset Shen et al. [2017] to train, and evaluate on a test set of 1000 sentences released by Li et al. [2018]. We also use the **Amazon** dataset of product reviews He and McAuley [2016]. We use the Yelp review dataset labelled for the **Gender** of the author, released by Prabhumoye et al. [2018] compiled from Reddy and Knight [2016]. For the **Political** slant task Prabhumoye et al. [2018], we use dataset released by Voigt et al. [2018].

## 5.3 Methodology

We are given non-parallel samples of sentences $\mathbf{X_1} = \{\mathbf{x}_1^{(1)} \dots \mathbf{x}_n^{(1)}\}$ and $\mathbf{X_2} = \{\mathbf{x}_1^{(2)} \dots \mathbf{x}_m^{(2)}\}$ from styles $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively. The objective of the task is to efficiently generate samples $\hat{\mathbf{X}}_\mathbf{1} = \{\hat{\mathbf{x}}_1^{(2)} \dots \hat{\mathbf{x}}_n^{(2)}\}$ in the target style $\mathcal{S}_2$, conditioned on samples in $\mathbf{X_1}$. For a style $\mathcal{S}_v$ where $v \in \{1, 2\}$, we begin by learning a set of phrases ($\Gamma_v$) which characterize the style $\mathcal{S}_v$. The presence of phrases from $\Gamma_v$ in a sentence $\mathbf{x}_i$ would associate the sentence with the style $\mathcal{S}_v$. For example, phrases like "pretty good" and "worth every penny" are characteristic of the "positive" style in the case of sentiment transfer task.

We propose a two staged approach where we first infer a sentence $z(\mathbf{x}_i)$ from $\mathbf{x}_i^{(1)}$ using a model, the tagger. The goal of the tagger is to ensure that the sentence $z(\mathbf{x}_i)$ is agnostic to the original style ($\mathcal{S}_1$) of the input sentence. Conditioned on $z(\mathbf{x}_i)$, we then generate the transferred sentence $\hat{\mathbf{x}}_i^{(2)}$ in the target style $\mathcal{S}_2$ using another model, the generator. The intermediate variable $z(\mathbf{x}_i)$ is also seen in other style-transfer methods. Shen et al. [2017], Prabhumoye et al. [2018], Yang et al. [2018b], Hu et al. [2017] transform the input $\mathbf{x}_i^{(v)}$ to a latent representation $z(\mathbf{x}_i)$ which (ideally) encodes the content present in $\mathbf{x}_i^{(v)}$ while being agnostic to style $\mathcal{S}_v$. In these cases $z(\mathbf{x}_i)$ encodes the input sentence in a continuous latent space whereas for us $z(\mathbf{x}_i)$ manifests in the surface form. The ability of our pipeline to generate observable intermediate outputs $z(\mathbf{x}_i)$ makes it somewhat more interpretable than those other methods.

We train two independent systems for the tagger & generator which have complimentary objectives. The former identifies the style attribute markers $a(x_i^{(1)})$ from source style $\mathcal{S}_1$ and either replaces them with a positional token called [TAG] or merely adds these positional tokens without removing any phrase from the input $x_i^{(1)}$. This particular capability of the model enables us to generate these tags in an input that is devoid of any attribute marker (i.e. $a(x_i^{(1)}) = \{\}$). This is one of the major differences from prior works which mainly focus on removing source style attributes and then replacing them with the target style attributes. It is especially critical for tasks like politeness transfer where the transfer takes place from a non-polite sentence. This is because in such cases we may need to add new phrases to the sentence rather than simply replace existing ones. The generator is trained to generate sentences $\hat{\mathbf{x}}_i^{(2)}$ in the target style by replacing these [TAG] tokens with stylistically relevant words inferred from target style $\mathcal{S}_2$. Even though we have non-parallel corpora, both systems are trained in a supervised fashion as sequence-to-sequence models with their own distinct pairs of inputs & outputs. To create parallel training data, we first estimate the style markers $\Gamma_v$ for a given style $\mathcal{S}_v$ & then use these to curate style free sentences with [TAG] tokens. Training data creation details are given in sections §5.3.2, §5.3.3.

Fig. 5.3 shows the overall pipeline of the proposed approach. In the first example $\mathbf{x}_1^{(1)}$, where there is no clear style attribute present, our model adds the [TAG] token in $z(\mathbf{x}_1)$, indicating that a target style marker should be generated in this position. On the contrary, in the second example, the terms "ok" and "bland" are markers of negative sentiment and hence the tagger has replaced them with [TAG] tokens in $z(\mathbf{x}_2)$. We can also see that the inferred sentence in both the cases is free of the original and target styles. The structural bias induced by this two staged approach is helpful in realizing an interpretable style free tagged sentence that explicitly encodes the content. In the following sections we discuss in detail the methodologies involved in (1) estimating the relevant attribute markers for a given style, (2) tagger, and (3) generator modules of our approach.

### 5.3.1 Estimating Style Phrases

Drawing from Li et al. [2018], we propose a simple approach based on n-gram tf-idfs to estimate the set $\Gamma_v$, which represents the style markers for style $v$. For a given corpus pair $\mathbf{X}_1, \mathbf{X}_2$ in styles $\mathcal{S}_1, \mathcal{S}_2$ respectively we first compute a probability distribution $p_1^2(w)$ over the n-grams $w$ present in both the corpora (Eq. 5.2). Intuitively, $p_1^2(w)$ is proportional to the probability of sampling an n-gram present in both $\mathbf{X}_1, \mathbf{X}_2$ but having a much higher tf-idf value in $\mathbf{X}_2$ relative to $\mathbf{X}_1$. This is how we define the impactful style markers for style $\mathcal{S}_2$.

$$\eta_1^2(w) = \frac{\frac{1}{m}\sum_{i=1}^{m} \text{tf-idf}(w, \mathbf{x}_i^{(2)})}{\frac{1}{n}\sum_{j=1}^{n} \text{tf-idf}(w, \mathbf{x}_j^{(1)})} \tag{5.1}$$

$$p_1^2(w) = \frac{\eta_1^2(w)^\gamma}{\sum_{w'} \eta_1^2(w')^\gamma} \tag{5.2}$$

where, $\eta_1^2(w)$ is the ratio of the mean tf-idfs for a given n-gram $w$ present in both $\mathbf{X}_1, \mathbf{X}_2$ with $|\mathbf{X}_1| = n$ and $|\mathbf{X}_2| = m$. Words with higher values for $\eta_1^2(w)$ have a higher mean tf-idf in $\mathbf{X}_2$ vs $\mathbf{X}_1$, and thus are more characteristic of $\mathcal{S}_2$. We further smooth and normalize $\eta_1^2(w)$ to get $p_1^2(w)$. Finally, we estimate $\Gamma_2$ by

$$\Gamma_2 = \{w : p_1^2(w) \geq k\}$$

In other words, $\Gamma_2$ consists of the set of phrases in $\mathbf{X}_2$ above a given style impact $k$. $\Gamma_1$ is computed similarly where we use $p_2^1(w), \eta_2^1(w)$.

### 5.3.2 Style Invariant Tagged Sentence

The tagger model (with parameters $\theta_t$) takes as input the sentences in $\mathbf{X}_1$ and outputs $\{z(\mathbf{x}_i) : \mathbf{x}_i^{(1)} \in \mathbf{X}_1\}$. Depending on the style transfer task, the tagger is trained to either (1) identify and replace style attributes $a(\mathbf{x}_i^{(1)})$ with the token tag [TAG] (replace-tagger) or (2) add the [TAG] token at specific locations in $\mathbf{x}_i^{(1)}$ (add-tagger). In both the cases, the [TAG] tokens indicate positions where the generator can insert phrases from the target style $\mathcal{S}_2$. Finally, we use the distribution $p_1^2(w)/p_2^1(w)$ over $\Gamma_2/\Gamma_1$ (§5.3.1) to draw samples of attribute-markers that would be replaced with the [TAG] token during the creation of training data.

The first variant, replace-tagger, is suited for a task like sentiment transfer where almost every sentence has some attribute markers $a(\mathbf{x}_i^{(1)})$ present in it. In this case the training data comprises of pairs where the input is $\mathbf{X}_1$ and the output is $\{z(\mathbf{x}_i) : \mathbf{x}_i^{(1)} \in \mathbf{X}_1\}$. The loss objective for replace-tagger is given by $\mathcal{L}_r(\theta_t)$ in Eq. 5.3.

$$\mathcal{L}_r(\theta_t) = -\sum_{i=1}^{|\mathbf{X}_1|} \log P_{\theta_t}(z(\mathbf{x}_i)|\mathbf{x}_i^{(1)}; \theta_t) \tag{5.3}$$

The second variant, add-tagger, is designed for cases where the transfer needs to happen from style neutral sentences to the target style. That is, $\mathbf{X}_1$ consists of style neutral sentences whereas $\mathbf{X}_2$ consists of sentences in the target style. Examples of such a task include the tasks of politeness transfer (introduced in this paper) and caption style transfer (used by Li et al. [2018]). In such cases, since the source sentences have no attribute markers to remove, the tagger learns to add [TAG] tokens at specific locations suitable for emanating style words in the target style.

The training data (Fig. 5.4) for the add-tagger is given by pairs where the input is $\{\mathbf{x}_i^{(2)} \backslash a(\mathbf{x}_i^{(2)}) : \mathbf{x}_i^{(2)} \in \mathbf{X}_2\}$ and the output is $\{z(\mathbf{x}_i) : \mathbf{x}_i^{(2)} \in \mathbf{X}_2\}$. Essentially, for the input we take samples

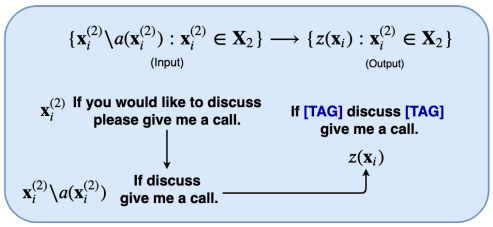



Figure 5.4: Creation of training data for add-tagger.

$\mathbf{x}_i^{(2)}$ in the target style $\mathcal{S}_2$ and explicitly remove style phrases $a(\mathbf{x}_i^{(2)})$ from it. For the output we replace the same phrases $a(\mathbf{x}_i^{(2)})$ with [TAG] tokens. As indicated in Fig. 5.4, we remove the style phrases "you would like to" and "please" and replace them with [TAG] in the output. Note that we only use samples from $\mathbf{X}_2$ for training the add-tagger; samples from the style neutral $\mathbf{X}_1$ are not involved in the training process at all. For example, in the case of politeness transfer, we only use the sentences labeled as "polite" for training. In effect, by training in this fashion, the tagger learns to add [TAG] tokens at appropriate locations in a style neutral sentence. The loss objective ($\mathcal{L}_a$) given by Eq. 5.4 is crucial for tasks like politeness transfer where one of the styles is poorly defined.

$$\mathcal{L}_a(\theta_t) = -\sum_{i=1}^{|\mathbf{X}_1|} \log P_{\theta_t}(z(\mathbf{x}_i)|\mathbf{x}_i^{(2)} \backslash a(\mathbf{x}_i^{(2)}); \theta_t) \tag{5.4}$$

### 5.3.3 Style Targeted Generation

The training for the generator model is complimentary to that of the tagger, in the sense that the generator takes as input the tagged output $z(\mathbf{x}_i)$ inferred from the source style and modifies the [TAG] tokens to generate the desired sentence $\hat{\mathbf{x}}_i^{(v)}$ in the target style $\mathcal{S}_v$.

$$\mathcal{L}(\theta_g) = -\sum_{i=1}^{|\mathbf{X}_v|} \log P_{\theta_g}(\mathbf{x}_i^{(v)}|z(\mathbf{x}_i); \theta_g) \tag{5.5}$$

The training data for transfer into style $\mathcal{S}_v$ comprises of pairs where the input is given by $\{z(\mathbf{x}_i) : x_i^{(v)} \in \mathbf{X}_v, v \in \{1, 2\}\}$ and the output is $\mathbf{X}_v$, i.e. it is trained to transform a style agnostic representation into a style targeted sentence. Since the generator has no notion of the original style and it is only concerned with the style agnostic representation $z(\mathbf{x}_i)$, it is convenient to disentangle the training for tagger & generator.

Finally, we note that the location at which the tags are generated has a significant impact on the distribution over style attributes (in $\Gamma_2$) that are used to fill the [TAG] token at a particular position. Hence, instead of using a single [TAG] token, we use a set of positional tokens $[\text{TAG}]_t$ where $t \in \{0, 1, \dots T\}$ for a sentence of length $T$. By training both tagger and generator with these positional $[\text{TAG}]_t$ tokens we enable them to easily realize different distributions of style attributes for different positions in a sentence. For example, in the case of politeness transfer, the

| | Politeness | | | | Gender | | | | Political | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc | BL-s | MET | ROU | Acc | BL-s | MET | ROU | Acc | BL-s | MET | ROU |
| CAE | **99.62** | 6.94 | 10.73 | 25.71 | 65.21 | 9.25 | 14.72 | 42.42 | 77.71 | 3.17 | 7.79 | 27.17 |
| BST | 60.75 | 2.55 | 9.19 | 18.99 | 54.4 | 20.73 | 22.57 | 55.55 | **88.49** | 10.71 | 16.26 | 41.02 |
| DRG | 90.25 | 11.83 | 18.07 | 41.09 | 36.29 | 22.9 | 22.84 | 53.30 | 69.79 | 25.69 | 21.6 | 51.8 |
| OURS | 89.50 | **70.44** | **36.26** | **70.99** | **82.21** | **52.76** | **37.42** | **74.59** | 87.74 | **68.44** | **45.44** | **77.51** |

Table 5.1: Results on the Politeness, Gender and Political datasets.

| | Yelp | | | | | Amazon | | | | | Captions | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc | BL-s | BL-r | MET | ROU | Acc | BL-s | BL-r | MET | ROU | Acc | BL-s | BL-r | MET | ROU |
| CAE | 72.1 | 19.95 | 7.75 | 21.70 | 55.9 | 78 | 2.64 | 1.68 | 9.52 | 29.16 | 89.66 | 2.09 | 1.57 | 9.61 | 30.02 |
| DRG | **88.8** | 36.69 | 14.51 | 32.09 | 61.06 | 52.2 | 57.07 | 29.85 | **50.16** | 79.31 | **95.65** | 31.79 | 11.78 | 32.45 | 64.32 |
| OURS | 86.6 | **47.14** | **19.76** | **36.26** | **70.99** | **66.4** | **68.74** | **34.80** | 45.3 | **83.45** | 93.17 | **51.01** | **15.63** | **43.67** | **79.51** |

Table 5.2: Results on the Yelp, Amazon and Captions datasets.

tags added at the beginning ($t = 0$) will almost always be used to generate a token like "Would it be possible ..." whereas for a higher $t$, $[\text{TAG}]_t$ may be replaced with a token like "thanks" or "sorry."

## 5.4 Experiments and Results

**Baselines** We compare our systems against three previous methods. DRG Li et al. [2018], Style Transfer Through Back-translation (BST) Prabhumoye et al. [2018], and Style transfer from non-parallel text by cross alignment Shen et al. [2017] (CAE). For DRG, we only compare against the best reported method, delete-retrieve-generate. For all the models, we follow the experimental setups described in their respective papers.

**Implementation Details** We use 4-layered transformers Vaswani et al. [2017] to train both tagger and generator modules. Each transformer has $4$ attention heads with a $512$ dimensional embedding layer and hidden state size. Dropout Srivastava et al. [2014] with p-value $0.3$ is added for each layer in the transformer. For the politeness dataset the generator module is trained with data augmentation techniques like random word shuffle, word drops/replacements as proposed by Im et al. [2017]. We empirically observed that these techniques provide an improvement in the fluency and diversity of the generations. Both modules were also trained with the BPE tokenization Sennrich et al. [2016] using a vocabulary of size $16000$ for all the datasets except for Captions, which was trained using $4000$ BPE tokens. The value of the smoothing parameter $\gamma$ in Eq. 5.2 is set to $0.75$. For all datasets except Yelp we use phrases with $p_1^2(w) \geq k = 0.9$ to construct $\Gamma_2$, $\Gamma_1$ (§5.3.1). For Yelp $k$ is set to $0.97$. During inference we use beam search (beam size=5) to decode tagged sentences and targeted generations for tagger & generator respectively. For the tagger, we re-rank the final beam search outputs based on the number of [TAG] tokens in the output sequence (favoring more [TAG] tokens).

**Automated Evaluation** Following prior work Li et al. [2018], Shen et al. [2017], we use automatic metrics for evaluation of the models along two major dimensions: (1) style transfer accuracy and (2) content preservation. To capture accuracy, we use a classifier trained on the nonparallel style corpora for the respective datasets (barring politeness). The architecture of the classifier is based on AWD-LSTM Merity et al. [2018] and a softmax layer trained via cross-entropy loss. We use the implementation provided by fastai.[5] For politeness, we use the classifier trained by Niu and Bansal [2018].[6] The metric of transfer accuracy **(Acc)** is defined as the percentage of generated sentences classified to be in the target domain by the classifier. The standard metric for measuring content preservation is BLEU-self **(BL-s)** Papineni et al. [2002] which is computed with respect to the original sentences. Additionally, we report the BLEU-reference **(BL-r)** scores using the human reference sentences on the Yelp, Amazon and Captions datasets Li et al. [2018]. We also report ROUGE **(ROU)** Lin [2004] and METEOR **(MET)** Denkowski and Lavie [2011] scores. In particular, METEOR also uses synonyms and stemmed forms of the words in candidate and reference sentences, and thus may be better at quantifying semantic similarities.

Table 5.1 shows that our model achieves significantly higher scores on BLEU, ROUGE and METEOR as compared to the baselines DRG, CAE and BST on the Politeness, Gender, and Political datasets. The BLEU score on the Politeness task is greater by $58.61$ points with respect to DRG. In general, CAE and BST achieve high classifier accuracies but they fail to retain the original content. The classifier accuracy on the generations of our model are comparable (within $1\%$) with that of DRG for the Politeness dataset.

In Table 5.2, we compare our model against CAE and DRG on the Yelp, Amazon, and Captions datasets. For each of the datasets our test set comprises 500 samples (with human references) curated by Li et al. [2018]. We observe an increase in the BLEU-reference scores by $5.25$, $4.95$ and $3.64$ on the Yelp, Amazon, and Captions test sets respectively. Additionally, we improve the transfer accuracy for Amazon by $14.2\%$ while achieving accuracies similar to DRG on Yelp and Captions. As noted by Li et al. [2018], one of the unique aspects of the Amazon dataset is the absence of similar content in both the sentiment polarities. Hence, the performance of their model is worse in this case. Since we don't make any such assumptions, we perform significantly better on this dataset.

While popular, the metrics of transfer accuracy and BLEU have significant shortcomings making them susceptible to simple adversaries. BLEU relies heavily on n-gram overlap and classifiers can be fooled by certain polarizing keywords. We test this hypothesis on the sentiment transfer task by a *Naive Baseline*. This baseline adds *"but overall it sucked"* at the end of the sentence to transfer it to negative sentiment. Similarly, it appends *"but overall it was perfect"* for transfer into a positive sentiment. This baseline achieves an average accuracy score of $91.3\%$ and a BLEU score of $61.44$ on the Yelp dataset. Despite high evaluation scores, it does not reflect a high rate of success on the task. In summary, evaluation via automatic metrics might not truly correlate with task success.

**Changing Content Words** Given that our model is explicitly trained to generate new content only in place of the TAG token, it is expected that a well-trained system will retain most of

[5] https://docs.fast.ai/

[6] This is trained on the dataset given by Danescu-Niculescu-Mizil et al. [2013].

| | Con | | Att | | Gra | |
|---|---|---|---|---|---|---|
| | DRG | Ours | DRG | Ours | DRG | Ours |
| Politeness | 2.9 | **3.6** | 3.2 | **3.6** | 2.0 | **3.7** |
| Gender | 3.0 | **3.5** | - | - | 2.2 | **2.5** |
| Political | 2.9 | **3.2** | - | - | 2.5 | **2.7** |
| Yelp | 3.0 | **3.7** | 3 | **3.9** | 2.7 | **3.3** |

Table 5.3: Human evaluation on Politeness, Gender, Political and Yelp datasets.

the non-tagged (content) words. Clearly, replacing content words is not desired since it may drastically change the meaning. In order to quantify this, we calculate the fraction of non-tagged words being changed across the datasets. We found that the non-tagged words were changed for only 6.9% of the sentences. In some of these cases, we noticed that changing non-tagged words helped in producing outputs that were more natural and fluent.

| Input | DRG Output | Our Model Output | Strategy |
|---|---|---|---|
| what happened to my personal station? | what happened to my mother to my co??? | could you please let me know what happened to my personal station? | Counterfactual Modal |
| yes, go ahead and remove it. | yes, please go to the link below and delete it. | yes, we can go ahead and remove it. | 1st Person Plural |
| not yet-i'll try this wkend. | not yet to say-i think this will be a <unk> long. | sorry not yet-i'll try to make sure this wk | Apologizing |
| please check on metromedia energy, | thanks again on the energy industry, | please check on metromedia energy, thanks | Mitigating please start |

Table 5.4: Qualitative Examples comparing the outputs from DRG and Our model for the Politeness Transfer Task

**Human Evaluation** Following Li et al. [2018], we select 10 unbiased human judges to rate the output of our model and DRG on three aspects: (1) content preservation **(Con)** (2) grammaticality of the generated content **(Gra)** (3) target attribute match of the generations **(Att)**. For each of these metrics, the reviewers give a score between 1-5 to each of the outputs, where 1 reflects a poor performance on the task and 5 means a perfect output. Since the judgement of signals that indicate gender and political inclination are prone to personal biases, we don't annotate these tasks for target attribute match metric. Instead we rely on the classifier scores for the transfer. We've used the same instructions from Li et al. [2018] for our human study. Overall, we evaluate both systems on a total of 200 samples for Politeness and 100 samples each for Yelp, Gender and Political.

Table 5.3 shows the results of human evaluations. We observe a significant improvement in content preservation scores across various datasets (specifically in Politeness domain) highlighting

the ability of our model to retain content better than DRG. Alongside, we also observe consistent improvements of our model on target attribute matching and grammatical correctness.

**Qualitative Analysis** We compare the results of our model with the DRG model qualitatively as shown in Table 5.4. Our analysis is based on the linguistic strategies for politeness as described in Danescu-Niculescu-Mizil et al. [2013]. The first sentence presents a simple example of the *counterfactual modal* strategy inducing *"Could you please"* to make the sentence polite. The second sentence highlights another subtle concept of politeness of *1st Person Plural* where adding *"we"* helps being indirect and creates the sense that the burden of the request is shared between speaker and addressee. The third sentence highlights the ability of the model to add *Apologizing* words like *"Sorry"* which helps in deflecting the social threat of the request by attuning to the imposition. According to the *Please Start* strategy, it is more direct and insincere to start a sentence with *"Please"*. The fourth sentence projects the case where our model uses *"thanks"* at the end to express gratitude and in turn, makes the sentence more polite. Our model follows the strategies prescribed in Danescu-Niculescu-Mizil et al. [2013] while generating polite sentences.[7]

**Ablations** We provide a comparison of the two variants of the tagger, namely the replace-tagger and add-tagger on two datasets. We also train and compare them with a *combined* variant.[8] We train these tagger variants on the Yelp and Captions datasets and present the results in Table 5.5. We observe that for Captions, where we transfer a factual (neutral) to romantic/humorous sentence, the add-tagger provides the best accuracy with a relatively negligible drop in BLEU scores. On the contrary, for Yelp, where both polarities are clearly defined, the replace-tagger gives the best performance. Interestingly, the accuracy of the add-tagger is $\approx 50\%$ in the case of Yelp, since adding negative words to a positive sentence or vice-versa neutralizes the classifier scores. Thus, we can use the add-tagger variant for transfer from a polarized class to a neutral class as well.

To check if the combined tagger is learning to perform the operation that is more suitable for a dataset, we calculate the fraction of times the combined tagger performs add/replace operations on the Yelp and Captions datasets. We find that for Yelp (a polar dataset) the combined tagger performs 20% more replace operations (as compared to add operations). In contrast, on the CAPTIONS dataset, it performs 50% more add operations. While the combined tagger learns to use the optimal tagging operation to some extent, a deeper understanding of this phenomenon is an interesting future topic for research. We conclude that the choice of the tagger variant is dependent on the characterstics of the underlying transfer task.

[7] We provide additional qualitative examples for other tasks in the supplementary material.

[8] Training of combined variant is done by training the tagger model on the concatenation of training data for add-tagger and replace-tagger.

| | Yelp | | Captions | |
|---|---|---|---|---|
| | Acc | BL-r | Acc | BL-r |
| Add-Tagger | 53.2 | 20.66 | **93.17** | 15.63 |
| Replace-Tagger | **86.6** | 19.76 | 84.5 | 15.04 |
| Combined | 72.5 | **22.46** | 82.17 | **18.5**1 |

Table 5.5: Comparison of different *tagger* variants for Yelp and Captions datasets

# Chapter 6

# Think about it! Improving Defeasible Reasoning by First Modeling the Question Scenario

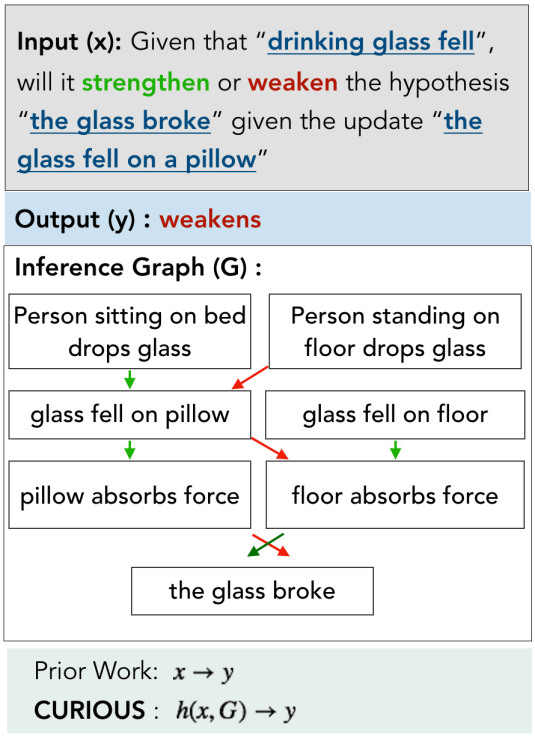



Figure 6.1: CURIOUS: Given a premise (*drinking glass fell*), defeasible reasoning seeks to answer whether new information that *the glass fell on a pillow* will strengthen or weaken the premise that *the glass broke*. CURIOUS improves defeasible reasoning by modeling the question scenario with a model-generated inference graph $G$. The graph $G$ includes new events (e.g., *glass fell on floor*) that contextualize the given premise, hypothesis, and update. The graph is encoded judiciously using our graph encoder $h(.)$, improving end-task performance.

Defeasible reasoning is the mode of reasoning where conclusions can be overturned by taking into account new evidence. Existing cognitive science literature on defeasible reasoning suggests that a person forms a *mental model* of the problem scenario before answering questions. Our research goal asks whether neural models can similarly benefit from envisioning the question scenario before answering a defeasible query. Our approach is, given a question, to have a model first create a graph of relevant influences, and then leverage that graph as an additional input when answering the question. Our system, CURIOUS, achieves a new state-of-the-art on three different defeasible reasoning datasets. This result is significant as it illustrates that performance can be improved by guiding a system to "think about" a question and explicitly model the scenario, rather than answering reflexively.[1]

## 6.1 Introduction

Defeasible inference is a mode of reasoning where additional information can modify conclusions Koons [2017]. Here we consider the specific formulation and challenge in Rudinger et al. [2020]: Given that some premise $p$ plausibly implies a hypothesis **H**, does new information that the situation is **S** weaken or strengthen the conclusion **H**? For example, consider the premise "The drinking glass fell" with a possible implication "The glass broke". New information that "The glass fell on a pillow" here *weakens* the implication.

We borrow ideas from the cognitive science literature that supports defeasible reasoning for humans with an *inference graph* [Pollock, 2009, 1987]. Inference graphs formulation in Madaan et al. [2021a], which we use in this paper, draws connections between the $p$, **H**, and **S** through mediating events. This can be seen as a *mental model* of the question scenario before answering the question Johnson-Laird [1983]. This paper asks the natural question: can modeling the question scenario with inference graphs help machines in defeasible reasoning?

Our approach is as follows. First, given a question, generate an inference graph describing important influences between question elements. Then, use that graph as an additional input when answering the defeasible reasoning query. Our proposed system, CURIOUS, comprises a graph generation module and a graph encoding module to use the generated graph for the query (Figure 6.2).

To generate inference graphs, we build upon past work that uses a sequence-to-sequence approach [Madaan et al., 2021a]. Briefly, Madaan et al. [2021a] train a T5-XXL [Raffel et al., 2020a] on a dataset of influence graphs [Tandon et al., 2019], and use the trained model to generate similar graphs for a new situation. However, our analysis revealed that the graphs can often be erroneous, and CURIOUS also includes an error correction module (Section 6.3.3) to generate higher quality inference graphs. This was important because we found that better graphs are more helpful in the downstream QA task.

The generated inference graph is then used for the QA task on three existing defeasible inference datasets from diverse domains, viz., $\delta$-SNLI (natural language inference) Bowman et al. [2015], $\delta$-SOCIAL (reasoning about social norms) Forbes et al. [2020], and $\delta$-ATOMIC (commonsense reasoning) Sap et al. [2019]. We show that the way the graph is encoded for input

[1]Code and data located at https://github.com/madaan/thinkaboutit

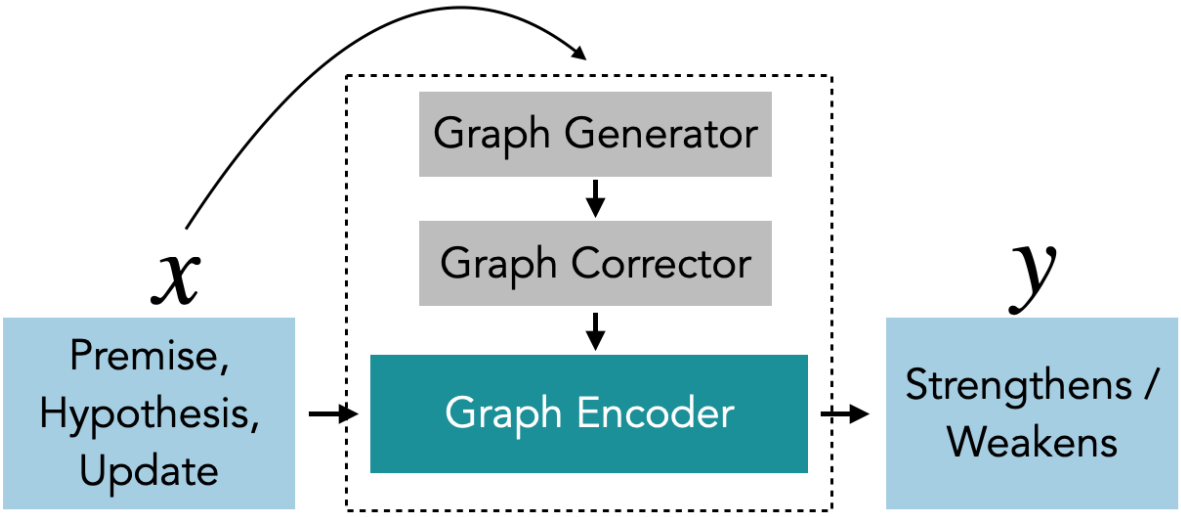



Figure 6.2: An overview of CURIOUS

is important. If we simply augment the question with the generated graphs, there are some gains on all datasets. However, the accuracy improves substantially across all datasets with a more judicious encoding of the graph-augmented question that accounts for interactions between the graph nodes. To achieve this, we use the mixture of experts approach Jacobs et al. [1991] to include a mixture of experts layers during encoding, enabling the ability to attend to specific nodes while capturing their interactions selectively.

In summary, our contribution is in drawing on the idea of an inference graph from cognitive science to show benefits in a defeasible inference QA task. Using an error correction module in the graph generation process, and a judicious encoding of the graph augmented question, CURIOUS achieves a new state-of-the-art over three defeasible datasets. This result is significant also because our work illustrates that guiding a system to "think about" a question before answering can improve performance.

## 6.2 Task

We use the defeasible inference task and datasets defined in Rudinger et al. [2020], namely given an input $\mathbf{x}$ = ($p$,**H**,**S**), predict the output $\mathbf{y} \in \{strengthens, weakens\}$, where $p$, **H**, and **S** are sentences describing a premise, hypothesis, and scenario respectively, and $y$ denotes whether $S$ strengthens/weakens the plausible conclusion that **H** follows from $p$, as described in Section 6.1.

## 6.3 Approach

Inspired by past results Madaan et al. [2021a] that humans found inference graphs useful for defeasible inference, we investigate whether neural models can benefit from envisioning the question scenario using an inference graph before answering a defeasible inference query.

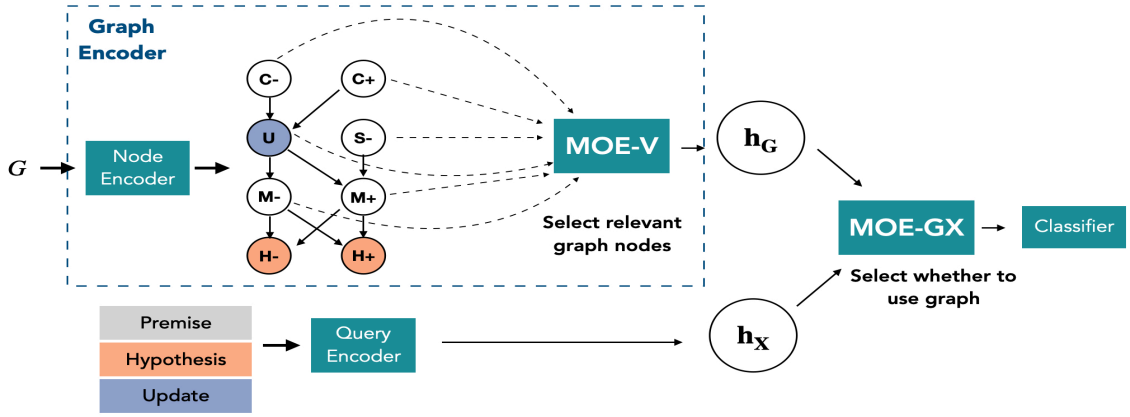



Figure 6.3: An overview of our method to perform graph-augmented defeasible reasoning using a hierarchical mixture of experts. First, **MOE-V** selectively pools the node representations to generate a representation $\mathbf{h_G}$ of the inference graph. Then, **MOE-GX** pools the query representation $\mathbf{h_x}$ and the graph representation generated by **MOE-V** to pass to the upstream classifier.

**Inference graphs** As inference graphs are central to our work, we give a brief description of their structure next. Inference graphs were introduced in philosophy by Pollock [2009] to aid defeasible reasoning for humans, and in NLP by Tandon et al. [2019] for a counterfactual reasoning task. We interpret the inference graphs as having four kinds of nodes Pollock [2009], Madaan et al. [2021a]:

i. **Contextualizers (C-, C+):** these nodes set the context around a situation and connect to the $p$.

ii. **Situations (S, S-):** these nodes are new situations that emerge which might overturn an inference.

iii. **Hypothesis (H-, H+):** Hypothesis nodes describe the outcome/conclusion of the situation.

iv. **Mediators (M-, M+):** Mediators are nodes that help bridge the knowledge gap between a situation and a hypothesis node by explaining their connection explicitly. These node can either act as a *weakener* or *strengthener*.

Each node in an influence graph is labeled with an event (a sentence or a phrase). The signs **-** and **+** capture the nature of the influence event node. Concrete examples are present in Figures 6.1, 6.4, and in Appendix E.4.

### 6.3.1 Overview of CURIOUS

Our system, CURIOUS, comprises three components, (i) a graph generator $\text{GEN}_{\text{init}}$, (ii) a graph corrector $\text{GEN}_{\text{corr}}$, (iii) a graph encoder (Figure 6.1). $\text{GEN}_{\text{init}}$ generates an inference graph from the input $\mathbf{x}$. We borrow the sequence to sequence approach of $\text{GEN}_{\text{init}}$ from Madaan et al. [2021a] without any architectural changes. However, we found that the resulting graphs can often be erroneous (which hurts task performance), so CURIOUS includes an error correction module $\text{GEN}_{\text{corr}}$ to generate higher quality inference graphs that are then judiciously encoded using the graph encoder. This encoded representation is then passed through a classifier to generate an end task label. The overall architecture is shown in Figure 6.2.

### 6.3.2 Graph generator

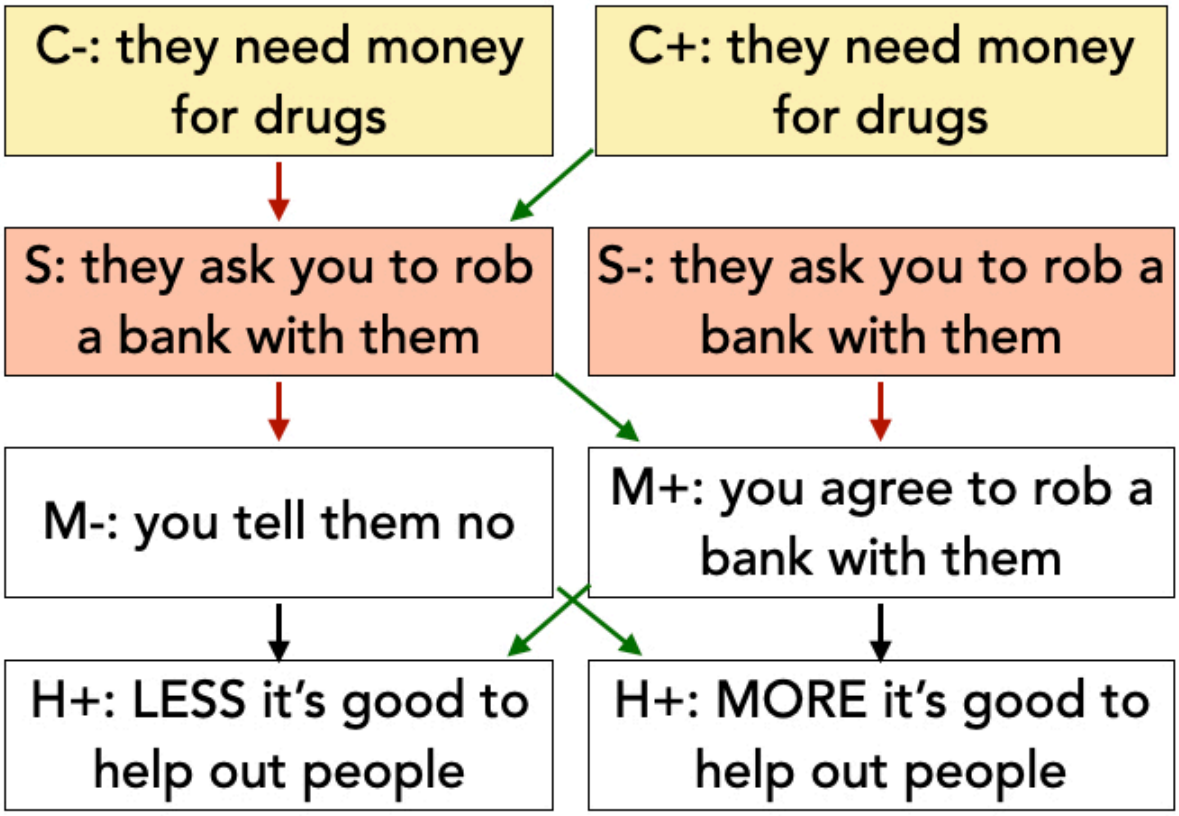



Figure 6.4: The graphs generated by $\text{GEN}_{\text{init}}$.The input graph has repetitions for nodes $\{C-, C+\}$ and $\{S, S-\}$. The corrected graph generated by $\text{GEN}_{\text{corr}}$ replaces the repetitions with meaningful labels.

As the initial graph generator, we use the method described in Madaan et al. [2021a] ($\text{GEN}_{\text{init}}$) to generate inference graphs for defeasible reasoning.[2] Their approach involves first training a graph-generation module, and then performing zero-shot inference on a defeasible query to obtain an inference graph. They obtain training data for the graph-generation module from WIQA dataset Tandon et al. [2019]. WIQA is a dataset of 2107 $(\mathbf{T}_i, \mathbf{G}_i)$ tuples, where $\mathbf{T}_i$ is the passage text that describes a process (e.g., waves hitting a beach), and $\mathbf{G}_i$ is the corresponding influence graph. The graph generator $\text{GEN}_{\text{init}}$ is trained as a seq2seq model, by setting `input` = [Premise] $\mathbf{T}_i$ | [Situation] $\mathbf{S}_i$ | [Hypothesis] $\mathbf{H}_i$, and `output` = $\mathbf{G}_i$. Note that $\mathbf{S}_i$ and $\mathbf{H}_i$ are nodes in the influence graph $\mathbf{G}_i$, allowing grounded generation. [Premise], [Situation], [Hypothesis] are special tokens used to demarcate the input.

[2]We use their publicly available code and data

### 6.3.3 Graph corrector

We found that 70% of the randomly sampled 100 graphs produced by $\text{GEN}_{\text{init}}$ (undesirably) had *repeated* nodes (an example of repeated nodes is in Figure 6.4). Repeated nodes introduce noise because they violate the semantic structure of a graph, e.g., in Figure 6.4, nodes C+ and C- are repeated, although they are expected to have opposite semantics. Higher graph quality yields better end task performance when using inference graphs (as we will show in §6.4.3)

To repair such problems, we train a graph corrector, $\text{GEN}_{\text{corr}}$, that takes as input $\mathbf{G}'$, and as output it gives a graph $\mathbf{G}^*$, with repetitions fixed. To train the model, we require ($\mathbf{G}'$, $\mathbf{G}^*$) examples, which we generate using a data augmentation technique described in the Appendix E.1. Because the nodes in the graph are from an open vocabulary, we then train a T5 sequence-to-sequence model Raffel et al. [2020c] with input = $\mathbf{G}'$ and output = $\mathbf{G}^*$. In summary, given a defeasible query $\mathbf{PHS}$, we generate a potentially incorrect initial graph $\mathbf{G}'$ using $\text{GEN}_{\text{init}}$. We then feed $\mathbf{G}'$ to $\text{GEN}_{\text{corr}}$ to obtain an improved graph $\mathbf{G}$.

### 6.3.4 Graph Encoder

For each defeasible query ($p$, $\mathbf{H}$, $\mathbf{S}$), we add the inference graph $\mathbf{G}$ from CURIOUS (the corrected graph from 6.3.3), to provide additional context for the query, as we now describe.

We concatenate the components ($p$, $\mathbf{H}$, $\mathbf{S}$) of the defeasible query into a single sequence of tokens $\mathbf{x} = (\mathbf{P}\|\mathbf{H}\|\mathbf{S})$, where $\|$ denotes concatenation. Thus, each sample of our graph-augmented binary-classification task takes the form $((\mathbf{x}, \mathbf{G}), \mathbf{y})$, where $\mathbf{y} \in \{\textit{strengthener}, \textit{weakener}\}$. Following Madaan et al. [2021a], we do not use edge labels and treat all the graphs as undirected graphs.

**Overview:** We first use a language model LMs to obtain a dense representation $\mathbf{h_x}$ for the defeasible query $\mathbf{x}$, and a dense representation $\mathbf{h_v}$ for each node $\mathbf{v} \in \mathbf{G}$. The node representations $\mathbf{h_v}$ are then pooled using a hierarchical mixture of experts (MOE) to obtain a graph representation $\mathbf{h_G}$. The query representation $\mathbf{h_x}$ and the graph representation $\mathbf{h_G}$ are combined to solve the defeasible task. We now provide details on obtaining $\mathbf{h_x}$, $\mathbf{h_v}$, $\mathbf{h_G}$.

#### Encoding the query and nodes

Let LMs be a pre-trained language model (in our case RoBERTa Liu et al. [2019]). We use $\mathbf{h_S} = \mathcal{L}(\mathbf{S}) \in \mathbb{R}^d$ to denote the dense representation of sequence of tokens $\mathbf{S}$ returned by the language model LMs. Specifically, we use the pooled representation of the beginning-of-sequence token <s> as the sequence representation.

We encode the defeasible query $\mathbf{x}$ and the nodes of the graph using LMs. Query representation is computed as $\mathbf{h_x} = \mathcal{L}(\mathbf{x})$, and we similarly obtain a matrix of node representations $\mathbf{h_V}$ by encoding each node $\mathbf{v}$ in $\mathbf{G}$ with LMs as follows:

$$\mathbf{h_V} = [\mathbf{h}_{v_1}; \mathbf{h}_{v_2}; \ldots; \mathbf{h}_{|\mathbf{V}|}] \tag{6.1}$$

where $\mathbf{h}_{v_i} \in \mathbb{R}^d$ refers to the dense representation for the $i^{th}$ node of $\mathbf{G}$ derived from LMs (i.e., $\mathbf{h}_{v_i} = \mathcal{L}(v_i)$), and $\mathbf{h_V} \in \mathbb{R}^{|\mathbf{V}| \times d}$ to refer to the matrix of node representations.

**Graph representations using MOE**

Recently, mixture-of-experts Jacobs et al. [1991], Shazeer et al. [2017], Fedus et al. [2021] has emerged as a promising method of combining multiple feature types. Mixture-of-experts (MOE) is especially useful when the input consists of multiple *facets*, where each facet has a specific semantic meaning. Previously, Gu et al. [2018], Chen et al. [2019] have used the ability of MOE to pool disparate features on low-resource and cross-lingual language tasks. Since each node in the inference graphs used by us plays a specific role in defeasible reasoning (contextualizer, situation node, or mediator), we take inspiration from these works to design a hierarchical MOE model Jordan and Xu [1995] to pool node representations $\mathbf{h_V}$ into a graph representation $\mathbf{h_G}$.

An MOE consists of $n$ expert networks $\mathbf{E_1}, \mathbf{E_2}, \ldots, \mathbf{E_n}$ and a gating network $\mathbf{M}$. Given an input $\mathbf{x} \in \mathbb{R}^d$, each expert network $\mathbf{E_i} : \mathbb{R}^d \rightarrow \mathbb{R}^k$ learns a transform over the input. The gating network $\mathbf{M} : \mathbb{R}^d \rightarrow \Delta^d$ gives the weights $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$ to combine the expert outputs for input $\mathbf{x}$. Finally, the output $\mathbf{y}$ is returned as a convex combination of the expert outputs:

$$\begin{aligned} \mathbf{p} &= \mathbf{M}(\mathbf{x}) \\ \mathbf{y} &= \sum_{i=1}^{n} p_i \mathbf{E_i}(\mathbf{x}) \end{aligned} \tag{6.2}$$

The output $\mathbf{y}$ can either be the logits for an end task Shazeer et al. [2017], Fedus et al. [2021] or pooled features that are passed to a downstream learner Chen et al. [2019], Gu et al. [2018]. The gating network $\mathbf{M}$ and the expert networks $\mathbf{E_1}, \mathbf{E_2}, \ldots, \mathbf{E_n}$ are trained end-to-end. During learning, the gradients to $\mathbf{M}$ train it to generate a distribution over the experts that favors the best expert for a given input. Appendix E.2 presents a further discussion on our MOE formulation and an analysis of the gradients.

**Hierarchical MOE for defeasible reasoning** Different parts of the inference graphs might help answer a query to a different degree. Further, for certain queries, graphs might not be helpful (and could even be distracting), and the model could rely primarily on the input query alone. This motivates a two-level architecture that can: (i) select a subset of the nodes in the graph and ii) selectively reason across the query and the graph to varying degrees.

Given these requirements, a hierarchical MOE Jordan and Jacobs [1994] model presents itself as a natural choice to model this task. The first MOE (**MOE-V**) creates a graph representation by taking a convex combination of the node representations. The second MOE (**MOE-GX**) then takes a convex-combination of the graph representation returned by **MOE-V** and query representation and passes it to an MLP for the downstream task.

- **MOE-V** consists of five node-experts and gating network to selectively pool node representations $\mathbf{h_v}$ to graph representation $\mathbf{h_G}$:

$$\begin{aligned} \mathbf{p} &= \mathbf{M}(\mathbf{h_V}) \\ \mathbf{h_G} &= \sum_{v \in \mathbf{V}} p_v E_v(v) \end{aligned} \tag{6.3}$$

- **MOE-GX** contains two experts (graph expert $E_\mathbf{G}$ and question expert $E_\mathbf{Q}$) and a gating network to combine the graph representation $\mathbf{h_G}$ returned by **MOE-GX** and the query representation

$\mathbf{h_x}$:

$$\mathbf{p} = \mathbf{M}([\mathbf{h_G}; \mathbf{h_Q}])$$
$$\mathbf{h_y} = E_{\mathbf{G}}(\mathbf{h_G}) + E_{\mathbf{Q}}(\mathbf{h_Q}) \tag{6.4}$$

$\mathbf{h_y}$ is then passed to a 1-layer MLP to perform classification. The gates and the experts in our MOE model are single-layer MLPs, with equal input and output dimensions for the experts.

## 6.4 Experiments

In this section, we empirically investigate if CURIOUS can improve defeasible inference by first modeling the question scenario using inference graphs. We also study the reasons for any improvements.

### 6.4.1 Experimental setup

| Dataset | Split | # Samples | Total |
|---|---|---|---|
| $\delta$-ATOMIC | train | 35,001 | 42,977 |
| | test | 4137 | |
| | dev | 3839 | |
| $\delta$-SOCIAL | train | 88,675 | 92,295 |
| | test | 1836 | |
| | dev | 1784 | |
| $\delta$-SNLI | train | 77,015 | 95,795 |
| | test | 9438 | |
| | dev | 9342 | |

Table 6.1: Number of samples in each dataset by split.

**Datasets** Our end task performance is measured on the three existing datasets for defeasible inference provided by Rudinger et al. [2020]:[3] $\delta$-ATOMIC, $\delta$-SNLI, $\delta$-SOCIAL (Table 6.1). These datasets exhibit substantial diversity because of their domains: $\delta$-SNLI (natural language inference), $\delta$-SOCIAL (reasoning about social norms), and $\delta$-ATOMIC (commonsense reasoning). Thus, it would require a general model to perform well across these diverse datasets.

**Baselines and setup** The previous state-of-the-art (SOTA) is the RoBERTa Liu et al. [2019] model presented in Rudinger et al. [2020], and we report the published numbers for this baseline. For an additional baseline, we directly use the initial inference graph $\mathbf{G}'$ generated by $\text{GEN}_{\text{init}}$,

[3] github.com/rudinger/defeasible-nli

and provide it to the model simply as a string (i.e., sequence of tokens; a simple, often-used approach). This baseline is called E2E-STR. We use the same hyperparameters as Rudinger et al. [2020], and add a detailed description of the hyperparameters in Appendix E.3. For all the QA experiments, we report the accuracy on the test set using the checkpoint with the highest accuracy on the development set. We use the McNemar's test McNemar [1947], Dror et al. [2018] and use $p < 0.05$ as a threshold for statistical significance. All the p-values are reported in Appendix E.7.

### 6.4.2 Results

Table 6.2 compares QA accuracy on these datasets without and with modeling the question scenario. The results suggest that we get consistent gains across all datasets, with $\delta$-SNLI gaining about 4 points. CURIOUS achieves a new state-of-the-art across three datasets, as well as now producing justifications for its answers with inference graphs.

| | $\delta$-ATOMIC | $\delta$-SNLI | $\delta$-SOCIAL |
|---|---|---|---|
| Prev-SOTA | 78.3 | 81.6 | 86.2 |
| E2E-STR | 78.8 | 82.2 | 86.7 |
| CURIOUS | **80.2*** | **85.6*** | **88.6*** |

Table 6.2: CURIOUS is better across all the datasets. This demonstrates that understanding the question scenario through generating an inference graph helps. * indicates statistical significance.

### 6.4.3 Understanding CURIOUS gains

In this section, we study the contribution of the main components of the CURIOUS pipeline.

#### Impact of graph corrector

We ablate the graph corrector module $\text{GEN}_{\text{corr}}$ in CURIOUS by directly supplying the output from $\text{GEN}_{\text{init}}$ to the graph encoder. Table 6.3 shows that this ablation consistently hurts across all the datasets. $\text{GEN}_{\text{corr}}$ provides 2 points gain across datasets. This indicates that better graphs lead to better task performance, assuming that $\text{GEN}_{\text{corr}}$ actually reduces the noise. Next, we investigate if $\text{GEN}_{\text{corr}}$ can produce more informative graphs.

**Do graphs corrected by $\text{GEN}_{\text{corr}}$ show fewer repetitions?** We evaluate the repetitions in the graphs produced by $\text{GEN}_{\text{init}}$ and $\text{GEN}_{\text{corr}}$ using two metrics: (i) repetitions per graph: average number of repeated nodes in a graph. (ii) % with repetitions: % of graphs with at least one repeated node.

Table 6.4 shows $\text{GEN}_{\text{corr}}$ does reduce repetitions by approximately 40% (2.11 to 1.25) per graph across all datasets, and also reduces the fraction of graphs with at least one repetition by 25.7% across.

| | δ-ATOMIC | δ-SNLI | δ-SOCIAL |
|---|---|---|---|
| $\mathbf{G}'$ | 78.5 | 83.8 | 88.2 |
| $G$ | **80.2*** | **85.6*** | **88.6** |

Table 6.3: Performance w.r.t. the graph used. $\mathbf{G}'$ is the initial graph from $\text{GEN}_{\text{init}}$, $G$ is the corrected graph from $\text{GEN}_{\text{corr}}$. Better graphs lead to better task performance. * indicates statistical significance.

| | Repetitions | $\text{GEN}_{\text{init}}$ | $\text{GEN}_{\text{corr}}$ |
|---|---|---|---|
| δ-ATOMIC | per graph | 2.05 | **1.26** |
| | % graphs | 72 | **48** |
| δ-SNLI | per graph | 2.09 | **1.18** |
| | % graphs | 73 | **46** |
| δ-SOCIAL | per graph | 2.2 | **1.32** |
| | % graphs | 75 | **49** |
| OVERALL | per graph | | **Δ -40%** |
| | % graphs | | **Δ -25.7%** |

Table 6.4: $\text{GEN}_{\text{corr}}$ reduces the inconsistencies in graphs. The number of repetitions per graph and percentage of graphs with some repetition, both improve.

### Impact of graph encoder

We experiment with two alternative approaches to graph encoding to compare our MOE approach by using the graphs generated by $\text{GEN}_{\text{corr}}$:

**1. Graph convolutional networks**: We follow the approach of Lv et al. [2020] who use GCN Kipf and Welling [2017] to learn rich node representations from graphs. Broadly, node representations are initialized by LMs and then refined using a GCN. Finally, multi-headed attention Vaswani et al. [2017] between question representation $\mathbf{h_x}$ and the node representations is used to yield $\mathbf{h_G}$. We add a detailed description of this method in Appendix E.8.

**2. String based representation**: Another popular approach Sakaguchi et al. [2021a] is to concatenate the string representation of the nodes, and then using LMs to obtain the graph representation $\mathbf{h_G} = \mathcal{L}(v_1 \| v_2 \| ..)$ where $\|$ denotes string concatenation.

Table 6.5 shows that MOE graph encoder improves end task performance significantly compared to the baseline.[4] In the following analysis, we study the reasons for these gains in-depth.

We hypothesize that GCN is less resistant to noise than MOE in our setting, thus causing a lower performance. The graphs augmented with each query are not human-curated and are instead generated by a language model in a zero-shot inference setting. Thus, the GCN style message passing might amplify the noise in graph representations. On the other hand, **MOE-V** first selects the most useful nodes to answer the query to form the graph representation $\mathbf{h_G}$. Further, **MOE-GX** can also decide to completely discard the graph representations, as it does in many cases where the true answer for the defeasible query is *weakens*.

[4]Appendix E.5 provides an analysis on time and memory requirements.

To further establish the possibility of message passing hampering the downstream task performance, we experiment with a GCN-MOE hybrid, wherein we first refine the node representations using a 2-layer GCN as used by Lv et al. [2020], and then pool the node representations using an MOE. We found the results to be about the same as ones we obtained with GCN (3rd-row Table 6.5), indicating that bad node representations are indeed the root cause for the bad performance of GCN. This is also supported by Shi et al. [2019] who found that noise propagation directly deteriorates network embedding and GCN is sensitive to noise.

Interestingly, graphs help the end-task even when encoded using a relatively simple STR based encoding scheme, further establishing their utility.

| | $\delta$-ATOMIC | $\delta$-SNLI | $\delta$-SOCIAL |
|---|---|---|---|
| STR | 79.5 | 83.1 | 87.2 |
| GCN | 78.9 | 82.4 | 88.1 |
| GCN + MOE | 78.7 | 84.3 | 87.8 |
| MOE | **80.2** | **85.6** | **88.6** |

Table 6.5: Contribution of MoE-based graph encoding compared with alternative graph encoding methods. The gains of MOE over GCN are statistically significant for all the datasets, and the gains over STR are significant for $\delta$-SNLI and $\delta$-SOCIAL.

### Detailed MOE analysis

We now analyze the two MoEs used in CURIOUS: (i) the MOE over the nodes (**MOE-V**), and (ii) the MOE over $\mathbf{G}$ and input $x$ (**MOE-GX**).

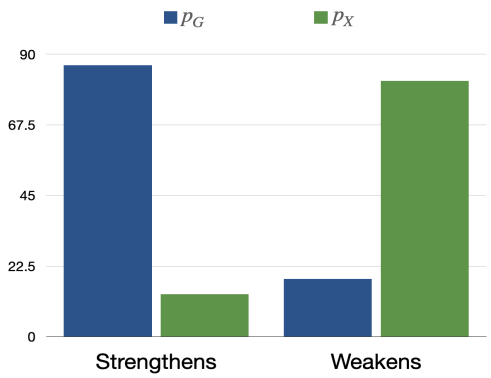



Figure 6.5: **MOE-GX** gate values for the classes strengthens and weakens, averaged over the three datasets.

**MOE-GX performs better for $y$ = strengthens:** Figure 6.5 shows that the graph makes a stronger contribution than the input, when the label is *strengthens*. In instances where the label is *weakens*, the gate of **MOE-GX** gives a higher weight to the question. This trend was present

across all the datasets. We conjecture that this happens because language models are tuned to generate events that happen rather than events that do not. In the case of a weakener, the nodes must be of the type *event1 leads to less of event2*, whereas language models are naturally trained for *event1 leads to event2*. Understanding this in-depth requires further investigation in the future.

**MOE-V relies more on specific nodes:** We study the distribution over the types of nodes and their contribution to MOE-V. Recall from Figure 6.3 that C- and C+ nodes are contextualizers that provide more background context to the question, and S- node is typically an inverse situation (i.e., inverse S), while M- and M+ are the mediator nodes leading to the hypothesis. Figure 6.6 shows that the situation node S- was the most important, followed by the contextualizer and the mediator. Notably, our analysis shows that mediators are less important for machines than they were for humans in the experiments conducted by Madaan et al. [2021a]. This is probably because humans and machines use different pieces of information.

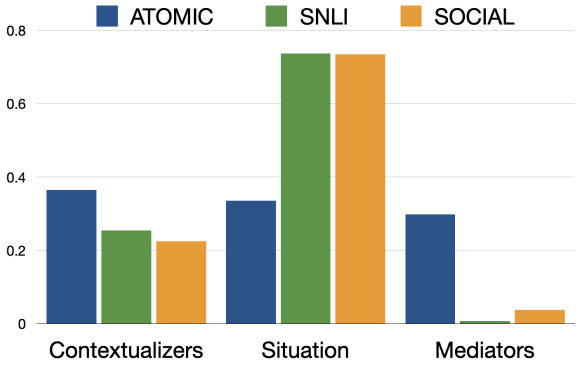



Figure 6.6: MOE-V gate values for the three datasets.

**MOE-V, MOE-GX have a peaky distribution:** A peaky distribution over the gate values implies that the network is judiciously selecting the right expert for a given input. We compute the average entropy of MOE-V and MOE-GX and found the entropy values to be 0.52 (max 1.61) for MOE-V, and 0.08 (max 0.69) for MOE-GX. The distribution of the gate values of MOE-V is relatively flat, indicating that specialization of the node experts might have some room for improvement (additional discussion in Appendix E.2). Analogous to scene graphs-based explanations in visual QA Ghosh et al. [2019], peaky distributions over nodes can be considered as an explanation through supporting nodes.

**MOE-V learns the node semantics:** The network learned the semantic grouping of the nodes (contextualizers, situation, mediators), which became evident when plotting the correlation between the gate weights. As Figure 6.7 shows, there is a strong negative correlation between the situation nodes and the context nodes, indicating that only one of them is activated at a time.

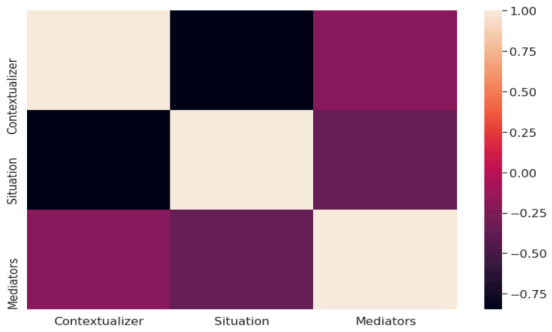



Figure 6.7: Correlation between probability assigned to each semantic type of the node by **MOE-V**

## 6.5 Summary and Conclusion

Cognitive science suggests that people form "mental models" of a situation to answer questions about it. Drawing on those ideas, we have presented a simple instantiation in which the situational model is an inference graph. Different from GCN-based models popular in graph learning, we use mixture-of-experts to pool graph representations. Our experiments show that MOE-based pooling can be a strong (both in terms of performance and explainability) alternative to GCN for graph-based learning for reasoning tasks. Our method establishes a new state-of-the-art on three defeasible reasoning datasets. Overall, our method shows that performance can be improved by guiding a system to "think about" a question and explicitly model the scenario, rather than answering reflexively.

# Part III

# Leveraging Structure During Inference

In Part 3 of this thesis, we address the challenges arising from the increasing size and complexity of large pre-trained language models (LLMs). As LLMs grow, training them becomes increasingly difficult due to their billions of parameters, often requiring infrastructure beyond the reach of all but a few large companies. On the other hand, these larger models exhibit remarkable few-shot inference capabilities, enabling them to perform a wide range of tasks with just a few examples. This often results in superior performance compared to smaller fine-tuned models.

Given the limitations imposed by the size of these models, the techniques presented in the previous chapters may not be directly applicable. However, we demonstrate that infusing structure remains advantageous even in the few-shot prompting regime. By incorporating structure during inference time, we can still achieve significant improvements in model performance, overcoming the constraints imposed by the model's size.

The chapters included in this part are:

1. Chapter 7 introduces CoCoGen, an approach for structured commonsense reasoning using large language models, which treats structured commonsense reasoning tasks as code generation tasks (EMNLP 2022).
2. Chapter 8 presents the Program-Aided Language models (PAL) approach, which leverages large language models for problem understanding and decomposition while outsourcing the solution step to a runtime (under submission at ICML 2023). This approach leads to improved performance in arithmetic and symbolic reasoning tasks.

These chapters highlight the value of incorporating structure during inference in the context of few-shot prompting, showcasing that even in the face of growing model complexity, we can still improve LLM performance by leveraging structured approaches.

# Chapter 7

# Language Models of Code are Few-Shot Commonsense Learners

We address the general task of *structured* commonsense reasoning: given a natural language input, the goal is to generate a *graph* such as an event or a reasoning-graph. To employ large language models (LMs) for this task, existing approaches "serialize" the output graph as a flat list of nodes and edges. Although feasible, these serialized graphs strongly deviate from the natural language corpora that LMs were pre-trained on, hindering LMs from generating them correctly. In this paper, we show that when we instead frame structured commonsense reasoning tasks as *code generation* tasks, pre-trained LMs of *code* are *better* structured commonsense reasoners than LMs of natural language, even when the downstream task does not involve source code at all. We demonstrate our approach across three diverse structured commonsense reasoning tasks. In all these *natural language* tasks, we show that using our approach, a *code* generation LM (CODEX) outperforms natural-LMs that are fine-tuned on the target task (e.g., T5) and other strong LMs such as GPT-3 in the few-shot setting.

## 7.1 Introduction

The growing capabilities of large pre-trained language models (LLM) for generating text have enabled their successful application in a variety of tasks, including summarization, translation, and question-answering [Wang et al., 2019, Raffel et al., 2020b, Brown et al., 2020b, Chowdhery et al., 2022b].

Nevertheless, while employing LLMs for natural language (NL) tasks is straightforward, a major remaining challenge is how to leverage LLMs for *structured commonsense reasoning*, including tasks such as generating event graphs [Tandon et al., 2019], reasoning graphs [Madaan et al., 2021b], scripts [Sakaguchi et al., 2021b], and argument explanation graphs [Saha et al., 2021]. Unlike traditional commonsense reasoning tasks such as reading comprehension or question answering, *structured* commonsense aims to generate structured output given a natural language input. This family of tasks relies on the natural language knowledge learned by the LLM, but it also requires complex structured prediction and generation.

To leverage LLM, existing structured commonsense generation models modify the *output*

*format* of a problem. Specifically, the structure to be generated (e.g., a graph or a table) is converted, or "serialized", into text. Such conversions include "flattening" the graph into a list of node pairs (Figure 7.1d), or into a specification language such as DOT [Figure 7.1c; Gansner et al., 2006].

While converting the structured output into text has shown promising results Rajagopal et al. [2022], Madaan and Yang [2021], LLM struggle to generate these "unnatural" outputs: LMs are primarily pre-trained on free-form text, and these serialized structured outputs strongly diverge from the majority of the pre-training data. Further, for natural language, semantically relevant words are typically found within a small span, whereas neighboring nodes in a graph might be pushed farther apart when representing a graph as a flat string.

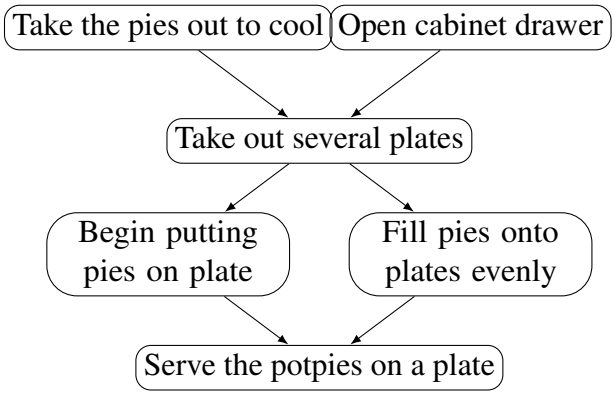
Thus, a language model which was trained on natural language text is likely to fail to capture the topology of the graph. Consequently, using LLM for graph generation typically requires a large amount of task-specific training data, and their generated outputs show structural errors and semantic inconsistencies, which need to be further fixed either manually or by using a secondary downstream model [Madaan et al., 2021d].

Despite these struggles, the recent success of large-language models of *code* [Code-LLMs; Chen et al., 2021d, Xu et al., 2022a] for tasks such as code generation from natural language Austin et al. [2021], Nijkamp et al. [2022a], code completion Fried et al. [2022a], and code translation Wang et al. [2021], show that Code-LLMs are able to perform complex reasoning on structured data such as programs. Thus, instead of forcing LLM of natural language (NL-LLMs) to be fine-tuned on structured commonsense data, an easier way to close the discrepancy between the pre-training data (free-form *text*) and the task-specific data (commonsense reasoning *graphs*) is to adapt LLMs that were pre-trained on *code* to structured commonsense reasoning in *natural language*.

Thus, our main insight is that *large language models of code are good structured commonsense reasoners*. Further, we show that Code-LLMs can be even better structured reasoners than NL-LLMs, when converting the desired output graph into a format similar to that observed in the code pre-training data. We call our method GPT-2: models of **Co**de for **Co**mmonsense **Gen**eration, and it is demonstrated in Figure 7.1.

Our contributions are as follows:

1. We highlight the insight that Code-LLMs are better structured commonsense reasoners than NL-LLMs, when representing the desired graph prediction as code.
2. We propose GPT-2: a method for leveraging LLMs of **co**de for structured **co**mmonsense **gen**eration.
3. We perform an extensive evaluation across three structured commonsense generation tasks and demonstrate that GPT-2 vastly outperforms NL-LLM, either fine-tuned or few-shot tested, while controlling for the number of downstream task examples.
4. We perform a thorough ablation study, which shows the role of data formatting, model size, and the number of few-shot examples.




(a) The script $\mathcal{G}$

```
class Tree:

  goal = "serve the potpies on a
      plate"

  def __init__(self):
    # nodes
    take_pies_out_to_cool = Node()
    open_cabinet_drawer = Node()
    take_out_several_plates = Node()
    ...
    # edges
    take_pies_out_to_cool.children =
        [take_out_several_plates]
    open_cabinet_drawer.children =
        [take_out_several_plates]
    ...
```

(b) $\mathcal{G}$ converted to Python code $\mathcal{G}_c$ using our approach

```
digraph G {
  begin -> take_pies_out_to_cool;
  begin -> open_cabinet_drawer;
  take_pies_out_to_cool ->
      take_out_several_plates;
  open_cabinet_drawer ->
      take_out_several_plates;
  take_out_several_plates ->
      begin_putting_pies_on_plates;
  begin_putting_pies_on_plates ->
      serve_potpies_on_plate;
  fill_pies_onto_plates_evenly ->
      serve_potpies_on_plate;
  serve_potpies_on_plate -> end;
}
```

(c) Straightforward encodings of the graph using the "DOT"

```
[
  (take_pies_out_to_cool,
      take_out_several_plates),
  (open_cabinet_drawer,
      take_out_several_plates),
  (take_out_several_plates,
      begin_putting_pies_on_plates),
  (take_out_several_plates,
      fill_pies_onto_plates_evenly),
  (begin_putting_pies_on_plates,
      serve_potpies_on_plate),
  (fill_pies_onto_plates_evenly,
      serve_potpies_on_plate),
  (serve_potpies_on_plate, end)
]
```

(d) Text format, or as a list of edges (node pairs)

Figure 7.1: An illustration of GPT-2 for the task of script generation. An input graph (7.1a) is typically represented using the DOT format (7.1c) or as a list of edges (7.1d), which allows modeling the graph using standard language models. These popular choices are sufficient in principle; however, these formats are loosely structured, verbose, and not common in text corpora, precluding language models from effectively generating them. In contrast, GPT-2 converts structures into Python code (7.1b), allowing to model them using large-scale language models of *code*.

## 7.2 GPT-2: Representing Commonsense structures with code

We focus on tasks of structured commonsense generation. Each training example for such tasks is in the form $(\mathcal{T}, \mathcal{G})$, where $\mathcal{T}$ is a text input, and $\mathcal{G}$ is the structure to be generated (typically a graph). The key idea of GPT-2 is transforming an output graph $\mathcal{G}$ into a semantically equivalent program $\mathcal{G}_c$ written in a general-purpose programming language. In this work, we chose Python due to its popularity in the training data of modern Code-LLMs [Xu et al., 2022a], but our approach is agnostic to the programming language. The code-transformed graphs are similar in their format to the pre-training data of Code-LLMs, and thus serve as easier to generalize training or few-shot examples than the original raw graph. GPT-2 uses Code-LLMs to generate $\mathcal{G}_c$ given $\mathcal{T}$, which we eventually convert back into the graph $\mathcal{G}$.

We use the task of script generation (PROSCRIPT, Figure 7.1) as a running example to motivate our method: script generation aims to create a script ($\mathcal{G}$) to achieve a given high-level goal ($\mathcal{T}$).

### 7.2.1 Converting $(\mathcal{T}, \mathcal{G})$ into Python code

We convert a $(\mathcal{T}, \mathcal{G})$ pair into a Python class or function. The general procedure involves adding the input text $\mathcal{T}$ in the beginning of the code as a class attribute or descriptive comment, and encoding the structure $\mathcal{G}$ using standard constructs for representing structure in code (e.g., hashmaps, object attributes) or function calls. The goal here is to compose Python code that represents a $(\mathcal{T}, \mathcal{G})$ pair, but retains the syntax and code conventions of typical Python code.

For example, for the script generation task, we convert the $(\mathcal{T}, \mathcal{G})$ pair into a `Tree` class (Figure 7.1b). The goal $\mathcal{T}$ is added as class attribute (`goal`), and the script $\mathcal{G}$ is added by listing the nodes and edges separately. We first instantiate the list of nodes as objects of class `Node`. Then, the edges are added as an attribute `children` for each node (Figure 7.1b). For example, we instantiate the node "*Take out several plates*" as `take_out_several_plates = Node()`, and add it as a child of the node `take_pies_out_to_cool`.

While there are multiple ways of representing a training example as a Python class, we found empirically that this relatively simple format is the most effective, especially with larger models. We analyze the choice of format and its connection with the model size in Section 7.4.

### 7.2.2 Few-shot prompting for generating $\mathcal{G}$

We focus on large-language models of the scale of CODEX [Chen et al., 2021b]. Due to their prohibitively expensive cost to fine-tune, these large models are typically used in a *few-shot prompting* mode. Few-shot prompting uses $k$ input-output examples $\{(x_i, y_i)\}_{i=1}^{k}$ to create an in-context prompt: $p = x_1 \oplus y_1 \cdot x_2 \oplus y_2 \cdot \ldots \cdot x_k \oplus y_k$, where $\oplus$ is a symbol that separates an input from its output, and $\cdot$ separates different examples.

A new (test) input $x$ is appended to the prompt $p$ (that is: $p \cdot x$), and $p \cdot x \oplus$ is fed to the model for completion. As found by Brown et al. [2020b], large language models show impressive few-shot capabilities in generating a completion $\hat{y}$ given the input $p \cdot x \oplus$. The main question is how to construct the prompt?

In all experiments in this work, the prompt $p$ consists of $k$ Python classes, each representing a $(\mathcal{T}, \mathcal{G}_c)$ pair. For example, for script generation, each Python class represents a goal $\mathcal{T}$ and a

script $\mathcal{G}_c$ from the training set. Given a new goal $\mathcal{T}$ for inference, a partial Python class (i.e., only specifying the goal) is created and appended to the prompt. Figure 7.2 shows such a partial class. Here, the code generation model is expected to complete the class by generating the definition for `Node` objects and their dependencies for the goal *make hot green tea*.

```
class Tree:
  goal = "make hot green tea."

  def __init__(self):
    # generate
```

Figure 7.2: GPT-2 uses a prompt consisting of $k$ (5-10) Python classes. During inference, the test input is converted to a partial class, as shown above, appended to the prompt, and completed by a code generation model such as CODEX.

In our experiments, we used CODEX [Chen et al., 2021b] and found that it nearly always generates syntactically valid Python. Thus, the generated code can be easily converted back into a graph and evaluated using the dataset's standard, original, metrics. Appendix G.6 lists sample prompts for each of the tasks we experimented with.

## 7.3 Evaluation

We experiment with three diverse structured commonsense generation tasks: (i) script generation (PROSCRIPT, Section 7.3.2), (ii) entity state tracking (PROPARA, Section 7.3.3), and (iii) explanation graph generation (EXPLAGRAPHS, Section 7.3.4) Dataset details are included in Appendix G.4. Despite sharing the general goal of structured commonsense generation, the three tasks are quite diverse in terms of the generated output and the kind of required reasoning.

### 7.3.1 Experimental setup

**Model** As our main Code-LLM for GPT-2, we experiment with the latest version of CODEX `code-davinci-002` from OpenAI[1] in few-shot prompting mode. While codex was initially released as a code generation model[2], OpenAI mentioned in a blog post released later that `code-davinci-002` was used to initialize `text-davinci-002` [Fu and Khot, 2022]. This suggests that code-based pre-training can boost a model's general reasoning abilities – a finding corroborated by recent research [Shao et al., 2024]. Also see [Fu and Khot, 2022] for a discussion on how pretraining on code could have assisted reasoning capabilities for large language models.

**Baselines** We experimented with the following types of baselines:

[1] As of June 2022

[2] https://openai.com/blog/openai-codex

| | BLEU | ROUGE-L | BLEURT | ISO | GED | Avg(d) | Avg($\|V\|$) | Avg($\|E\|$) |
|---|---|---|---|---|---|---|---|---|
| $\mathcal{G}$ (reference graph) | | - | - | 1.00 | 0.00 | 1.84 | 7.41 | 6.80 |
| T5 (fine-tuned) | 23.80 | 35.50 | -0.31 | 0.51 | 1.89 | **1.79** | 7.46 | **6.70** |
| CURIE (15) | 11.40 | 27.00 | -0.41 | 0.15 | 3.92 | 1.47 | 8.09 | 6.16 |
| DAVINCI (15) | 23.11 | 36.51 | -0.27 | **0.64** | **1.44** | 1.74 | 7.58 | 6.59 |
| GPT-2 (15) | **25.24** | **38.28** | **-0.26** | 0.53 | 2.10 | **1.79** | **7.44** | **6.70** |

Table 7.1: Semantic and structural metrics for the script generation task on PROSCRIPT. T5 is fine-tuned on the entire dataset, while the few-shot models (CURIE, DAVINCI, CODEX) use 15 examples in the prompt.

1. **Text few-shot:** Our hypothesis is that code-generation models can be repurposed to generate structured output better. Thus, natural baselines for our approach are NL-LLMs – language models trained on natural language corpus. We experiment with the latest versions of CURIE (`text-curie-001`) and DAVINCI (`text-davinci-002`), the two GPT-3 based models by OpenAI [Brown et al., 2020b]. For both these models, the prompt consists of $(\mathcal{T}, \mathcal{G})$ examples, where $\mathcal{G}$ is simply flattened into a string (as in Figure 7.1c). DAVINCI is estimated to be much larger in size than CURIE, as our experiments also reveal (Appendix G.1). DAVINCI, popularly known as GPT-3, is the strongest text-generation model available through OpenAI APIs.[3]
2. **Fine-tuning:** we fine-tune a T5-large model for EXPLAGRAPHS, and use the results from Sakaguchi et al. [2021b] on T5-xxl for PROSCRIPT tasks. In contrast to the few-shot setup where the model only has access to a few examples, fine-tuned models observe the *entire* training data of the downstream task.

**Choice of prompt** We created the prompt $p$ by randomly sampling $k$ examples from the training set. As all models have a bounded input size (e.g., 4096 tokens for CODEX `code-davinci-002` and 4000 for GPT-3 `text-davinci-002`), the exact value of $k$ is task dependent: more examples can fit in a prompt in tasks where $(\mathcal{T}, \mathcal{G})$ is short. In our experiments, $k$ varies between 5 and 30, and the GPT-3 baseline is always fairly given the same prompts as CODEX. To control for the variance caused by the specific examples selected into $p$, we repeat each experiment with at least 3 different prompts, and report the average. We report the mean and standard deviations in Appendix G.9.

**GPT-2:** We use GPT-2 to refer to setups where a CODEX is used with a Python prompt. In Section 7.4, we also experiment with dynamically creating a prompt for each input example, using a NL-LLMs with code prompts, and using Code-LLMs with textual prompts.

[3] https://beta.openai.com/docs/models/gpt-3

| | Method | *prec* | *rec* | $F_1$ |
|---|---|---|---|---|
| fine-tuned | T5 (100) | 52.26 | 52.91 | 51.89 |
| | T5 (1k) | 60.55 | 61.24 | 60.15 |
| | T5 (4k) | **75.71** | **75.93** | **75.72** |
| few-shot | CURIE (15) | 10.19 | 11.61 | 10.62 |
| | DAVINCI (15) | 50.62 | 49.30 | 48.92 |
| | GPT-2 (15) | **57.34** | **55.44** | **56.24** |

Table 7.2: Precision, recall, and $F_1$ for PROSCRIPT edge-prediction task. GPT-2 with 15 samples outperforms strong few-shot models, and T5 trained on 100 samples.

### 7.3.2 Script generation: PROSCRIPT

Given a high-level goal (e.g., *bake a cake*), the goal of script generation is to generate a graph where each node is an action, and edges capture dependency between the actions (Figure 7.1a). We use the PROSCRIPT [Sakaguchi et al., 2021b] dataset, where the scripts are directed acyclic graphs, which were collected from a diverse range of sources including ROCStories [Mostafazadeh et al., 2016], Descript [Wanzare et al., 2016], and Virtual home [Puig et al., 2018].

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a script for a high-level goal $\mathcal{T}$ with node and edge sets $\mathcal{V}$ and $\mathcal{E}$, respectively. Following Sakaguchi et al. [2021b], we experiment with two sub-tasks: (i) **script generation:** generating the entire script $\mathcal{G}(\mathcal{V}, \mathcal{E})$ given a goal $\mathcal{T}$, and (ii) **edge prediction:** predicting the edge set $\mathcal{E}$ **given** the nodes $\mathcal{V}$ and the goal $\mathcal{T}$.

Figure 7.1 shows an input-output example from PROSCRIPT, and our conversion of the graph into Python code: we convert each node $v \in \mathcal{V}$ into an instance of a `Node` class; we create the edges by adding `children` attribute for each of the nodes. Additional examples are present in Figure 39

To represent a sample for edge prediction, we list the nodes in a random order (specified after the comment `# nodes` in Figure 7.1b). The model then completes the class by generating the code below the comment `# edges`.

**Script Generation metrics** We denote the script that was generated by the model as $\hat{\mathcal{G}}$, and evaluate $\hat{\mathcal{G}}$ vs. $\mathcal{G}$ for both semantic and structural similarity. To evaluate semantic similarity, we use BLEU, ROUGE-L, and the learned metric BLEURT to determine the content overlap. Following Sakaguchi et al. [2021b], we use the following metrics for structural evaluation of generated scripts:

- Graph edit distance (GED): the number of required edits (node/edge removal/additions) to transform $\hat{\mathcal{G}}$ to $\mathcal{G}$ Abu-Aisheh et al. [2015];
- Graph isomorphism [ISO; Cordella et al., 2001]: determines whether $\hat{\mathcal{G}}$ and $\mathcal{G}$ are isomorphic based on their structure, disregarding the textual content of nodes;
- Graph size: average number of nodes and edges, $(|\mathcal{G}(V)|, |\mathcal{G}(E)|, |\hat{\mathcal{G}}(V)|, |\hat{\mathcal{G}}(V))$ and the average degree (d($\mathcal{G}(V)$)), where the high-level goal is for $\hat{\mathcal{G}}$ to have as close measures to $\mathcal{G}$ as possible.

**Edge Prediction metrics** For the edge prediction task, the set of nodes is given, and the goal is to predict the edges between them.

Following Sakaguchi et al. [2021b], we measure precision, recall, and $F_1$ comparing the true and predicted edges. Specifically, $p = \frac{|\boldsymbol{E} \cap \hat{\boldsymbol{E}}|}{|\hat{\boldsymbol{E}}|}$, $r = \frac{|\boldsymbol{E} \cup \hat{\boldsymbol{E}}|}{|\boldsymbol{E}|}$, and $F_1 = \frac{2pr}{p+r}$.

| Action | Entity | | |
|---|---|---|---|
| | water | light | CO2 |
| Initial states | soil | sun | - |
| Roots absorb water from soil | roots | sun | ? |
| The water flows to the leaf | leaf | sun | ? |

```
def main():
    # init
    # roots absorb water from soil
    # the water flows to the leaf
    # state_0 tracks the location/state water
    # state_1 tracks the location/state light
    # state_2 tracks the location/state CO2
    def init():
        state_0 = "soil"
        state_1 = "sun"
        state_2 = None
    def roots_absorb_water_from_soil():
        state_0 = "roots"
        state_1 = "sun"
        state_2 = "UNK"
    def water_flows_to_leaf():
        state_0 = "leaf"
        state_1 = "sun"
        state_2 = "UNK"
```

Figure 7.3: A PROPARA example (left) and its corresponding Python code (right). We use a string to represent a concrete location (e.g., `soil`), `UNK` to represent an unknown location, and `None` to represent non-existence.

**Results** Table 7.1 shows the results for script generation. The main results are that GPT-2 (based on CODEX), with just 15 prompt examples, outperforms the fine-tuned model T5 which has been fine-tuned on *all* 3500 samples. Further, GPT-2 outperforms the few-shot NL-LM CURIE across all semantic metrics and structural metrics. GPT-2 outperforms DAVINCI across all semantic metrics, while DAVINCI performs slightly better in two structural metrics.

Table 7.2 shows the results for edge prediction: GPT-2 significantly outperforms the NL-LLMs CURIE and DAVINCI. When comparing with T5, which was fine-tuned, GPT-2 with only 15 examples outperforms the fine-tuned T5 which was fine-tuned on 100 examples. The impressive performance in the edge-generation task allows us to highlight the better ability of GPT-2 in capturing structure, while factoring out all models' ability to generate the NL content.

### 7.3.3 Entity state tracking: PROPARA

The text inputs $\mathcal{T}$ of entity state tracking are a sequence of actions in natural language about a particular topic (e.g., photosynthesis) and a collection of entities (e.g., water). The goal is to predict the state of each entity after the executions of an action. We use the PROPARA dataset Dalvi et al. [2018] as the test-bed for this task.

| Model | $prec$ | $rec$ | $F_1$ |
|---|---|---|---|
| CURIE | **95.1** | 22.3 | 36.1 |
| DAVINCI | 75.5 | 47.1 | 58.0 |
| GPT-2 | 80.0 | **53.6** | **63.0** |

Table 7.3: 3-shots results on PROPARA. All numbers are averaged among five runs with different randomly sampled prompts. GPT-2 significantly outperforms CURIE and DAVINCI.

We construct the Python code $\mathcal{G}_c$ as follows, and an example is shown in Figure 7.3. First, we define the `main` function and list all $n$ actions as comments inside the `main` function. Second, we create $k$ variables named as `state_k` where $k$ is the number of participants of the topic. The semantics of each variable is described in the comments as well. Finally, to represent the state change after each step, we define $n$ functions where each function corresponds to an action. We additionally define an `init` function to represent the initialization of entity states. Inside each function, the value of each variable tells the state of the corresponding entity after the execution of that action. Given a new test example where only the actions and the entities are give, we construct the input string until the `init` function, and we append it to the few-shot prompts for predictions.

**Metrics** We follow Dalvi et al. [2018] and measure precision, recall and $F_1$ score of the predicted entity states. We randomly sampled three examples from the training set as the few-shot prompt.

**Results** As shown in Table 7.3, GPT-2 achieves a significantly better $F_1$ score than DAVINCI. Across the five prompts, GPT-2 achieves 5.0 higher $F_1$ than DAVINCI on average. In addition, GPT-2 yields stronger performance than CURIE, achieving $F_1$ of 63.0, which is 74% higher than CURIE (36.1).[4]

In PROPARA, GPT-2 will be ranked $6^{th}$ on the leaderboard.[5] However, all the methods above GPT-2 require fine-tuning on the entire training corpus. In contrast, GPT-2 uses only *3 examples* in the prompt and has a gap of less than 10 $F_1$ points vs. the current state-of-the-art [Ma et al., 2022]. In the few-shot settings, GPT-2 is state-of-the-art in PROPARA.

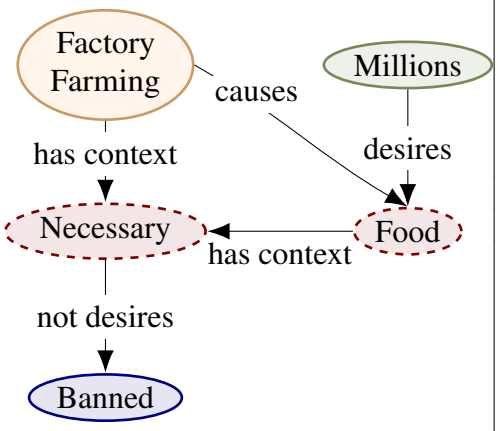
### 7.3.4 Argument graph generation: EXPLAGRAPHS

Given a belief (e.g., *factory farming should not be banned*) and an argument (e.g., *factory farming feeds millions*), the goal of this task is to generate a graph that uses the argument to either *support* or *counter* the belief [Saha et al., 2021]. The text input to the task is thus a tuple of (*belief*, *argument*, *"supports"/"counters"*), and the structured output is an explanation graph (Figure 7.4).

We use the EXPLAGRAPHS dataset for this task [Saha et al., 2021]. Since we focus on generating the argument graph, we take the stance as given and use the stance that was predicted

[4] CURIE often failed to produce output with the desired format, and thus its high precision and low recall.

[5] As of 10/11/2022, https://leaderboard.allenai.org/propara/submissions/public




```python
class ExplanationDAG:

  def __init__(self):
    belief = "factory farming should not be
        banned."
    argument = "Factory farming feeds
        millions."
    stance = "support"

    # Edges
    begin = ["factory farming", "millions"]
    add_edge("factory farming", "causes",
        "food")
    add_edge("factory farming", "has context",
        "necessary")
    add_edge("food", "has context",
        "necessary")
    add_edge("necessary", "not desires",
        "banned")
    add_edge("millions", "desires", "food")
```

Figure 7.4: An explanation graph (left) and the corresponding Python code (right)

by a stance prediction model released by Saha et al..

To convert an EXPLAGRAPHS to Python, the belief, argument, and stance are instantiated as string variables. Next, we define the graph structure by specifying the edges. Unlike PROSCRIPT, the edges in EXPLAGRAPHS are typed. Thus, each edge is added as an `add_edge(source, edge_type, destination)` function call. We also list the starting nodes in a list instantiated with a `begin` variable (Figure 7.4). Given a test example, we construct the input until the line of `# Edges` and let a model complete the remaining.

**Metrics** We use the metrics defined by Saha et al. [2021] (see Section 6 of Saha et al. [2021] for a detailed description of the mechanisms used to calculate these metrics):

- Structural accuracy (StCA): fraction of graphs that are connected DAGs with two concepts

| | | StCA (↑) | SeCA (↑) | G-BS (↑) | GED (↓) | EA (↑) |
|---|---|---|---|---|---|---|
| fine-tuned | T5 (150) | 12.56 | 6.03 | 9.54 | 91.06 | 7.77 |
| | T5 (1500) | 38.19 | 21.86 | 29.37 | 73.09 | 23.41 |
| | T5 (2500) | 43.22 | **29.65** | 33.71 | 69.14 | **26.38** |
| few-shot | CURIE (30) | 5.03 | 1.26 | 3.95 | 96.74 | 2.60 |
| | DAVINCI (30) | 23.62 | 10.80 | 18.46 | 83.83 | 11.84 |
| | GPT-2 (30) | **45.20** | **23.74** | **34.68** | **68.76** | **23.58** |

Table 7.4: Results for EXPLAGRAPHS (eval split). GPT-2 with only 30 examples outperforms the T5 model which was fine-tuned on 1500 examples, across all metrics.

| | EXPLAGRAPHS | | PROSCRIPT (edge-prediction) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | StCA (↑) | SeCA (↑) | G-BS (↑) | GED (↓) | EA (↑) | $p$ | $r$ | $F_1$ |
| DAVINCI + text | **33.16** | 7.14 | 25.91 | 77.45 | 15.9 | 43.06 | 41.52 | 43.06 |
| DAVINCI + code | 33.00 | **15.37** | **26.15** | **76.91** | **16.68** | **50.62** | **48.27** | **49.3** |
| CODEX + text | 38.02 | 18.23 | 29.46 | 73.68 | 19.54 | 45.31 | 43.95 | 44.47 |
| GPT-2 (CODEX + code) | **45.20** | **23.74** | **34.68** | **68.76** | **23.58** | **57.34** | **55.44** | **56.52** |

Table 7.5: Teasing apart the contributions of a code generation model and a structured prompt. The experiments show that both are helpful. DAVINCI, a text generation model, shows marginal improvements with a code prompt (top two rows). Similarly, CODEX, a code generation model, significantly benefits from a code prompt. Overall, CODEX with code prompt performs better than the alternatives, across all metrics.

each from belief and the argument.

- Semantic correctness (SeCA): a learned metric that evaluates if the correct stance is inferred from a (belief, graph) pair.
- G-BERTScore (G-BS): measures BERTscore- [Zhang et al., 2020a] based overlap between generated and reference edges .
- GED (GED): avg. edits required to transform the generated graph to the reference graph.
- Edge importance accuracy (EA): measures the importance of each edge in predicting the target stance. A high EA implies that each edge in the generated output contains unique semantic information, and removing any edge will hurt.

**Results** Table 7.4 shows that GPT-2 with only 30 examples outperforms the T5 model that was fine-tuned using 1500 examples, across all metrics. Further, GPT-2 outperforms the NL-LLMs DAVINCI and CURIE with a text-prompt across all metrics by about 50%-100%.

## 7.4 Analysis

In this section, we analyze the effect of three important components of GPT-2: (i) the contributions of Code-LLMs and structured prompt $\mathcal{G}_c$; (ii) the selection of examples in the in-context prompt; and (iii) the design of the Python class.

**Structured Prompts vs. Code-LLMs** Which component is more important, using a Code-LLMs or the structured formatting of the input as code? To answer this, we experimented with a text prompt with a primarily Code-LLM CODEX, and a code prompt with an NL-LLM, DAVINCI. DAVINCI was initialized from CODEX. However, unlike Codex, which focused on generating code, Davinci was specifically trained for following instructions in natural language [Fu and Khot, 2022]. Table 7.5 shows that both contributions are indeed important: performance improves for the NL-LLM DAVINCI both when we use a code prompt, *and* when we use a Code-LLM.

However when using both a Code-LLM and a code prompt – the improvement is greater than the sum of each of these solely.

**Dynamic prompt selection** The prompts for all experiments in Section 7.3 were created by *random* sampling of examples from the training set. Specifically, a set of $k$ $(\mathcal{T}, \mathcal{G})$ pairs are sampled and concatenated into a prompt $p$, which we used for inference over all examples $x_{test}$ in the test set. As an alternative to creating prompts, there is now a growing interest in customizing the in-context examples each example $x_{test}$. Popular techniques typically train a retriever, which is used to fetch the closest examples [Liu et al., 2022a, Rubin et al., 2022, Poesia et al., 2022]. We also experimented with such *dynamic* creation of the prompt, that depends on the particular test example. Specifically, following Poesia et al. [2022], we performed knowledge similarity tuning (KST): we trained a retriever model to retrieve the $k$ closest examples for a given input.

| Setup | p | r | $F_1$ |
|---|---|---|---|
| GPT-2 | 57.34 | 55.44 | 56.52 |
| GPT-2 + KST | **67.11** | **64.57** | **65.71** |

Table 7.6: Our retrieval mechanism is highly effective for edge prediction: the closest examples are from similar domains and the model is able to leverage the information for better performance.

The results indicate that the efficacy of dynamic prompts depends on both the training data and task. In the edge-prediction sub-task of PROSCRIPT, edges between events in similar scripts are helpful, and Table 7.6 shows that the model was able to effectively leverage this information. In the script generation sub-task of PROSCRIPT, Table 34 shows that KST provides gains as well (Appendix G.2).

In EXPLAGRAPHS, we observed that the training data had multiple examples which were nearly identical, and thus dynamically created prompts often included such duplicate examples, effectively reducing diversity and prompt size (Table 35).

**Python Formatting** We performed an extensive study of the effect of the Python format on the downstream task performance in Appendix G.7. We find that: (i) there are no clear task-agnostic Python class designs that work uniformly well; and that (ii) larger models are less sensitive to prompt (Python class) design. In general, our approach benefits the most from code formats that as similar as possible to the conventions of typical code.

**Human evaluation** We conduct human evaluation of the graphs generated by GPT-2 and DAVINCI to supplement automated metrics. The results (Appendix G.3) indicate that human evaluation is closely correlated with the automated metrics: for EXPLAGRAPHS, graphs generated by GPT-2 are found to be more relevant and correct. For PROSCRIPT generation, both DAVINCI and GPT-2 have complementary strengths, but GPT-2 is generally better in terms of relevance.

# Chapter 8

# PAL: Program-aided Language Models

Large language models (LLMs) have demonstrated an impressive ability to perform arithmetic and symbolic reasoning tasks, when provided with a few examples at test time ("few-shot prompting"). Much of this success can be attributed to prompting methods such as "chain-of-thought", which employ LLMs for both *understanding* the problem description by decomposing it into steps, as well as *solving* each step of the problem. While LLMs seem to be adept at this sort of step-by-step decomposition, LLMs often make logical and arithmetic mistakes in the solution part, even when the problem is decomposed correctly. In this paper, we present Program-Aided Language models (PAL): a novel approach that uses the LLM to read natural language problems and generate *programs* as the intermediate reasoning steps, but offloads the *solution* step to a runtime such as a Python interpreter. With PAL, decomposing the natural language problem into runnable steps remains the only learning task for the LLM, while solving is delegated to the interpreter. We demonstrate this synergy between a neural LLM and a symbolic interpreter across 13 mathematical, symbolic, and algorithmic reasoning tasks from BIG-Bench Hard and others. In all these *natural language* reasoning tasks, generating code using an LLM and reasoning using a Python interpreter leads to more accurate results than much larger models. For example, PAL using CODEX achieves state-of-the-art few-shot accuracy on Math Reasoning, surpassing PaLM-540B which uses chain-of-thought by absolute 15% top-1.[1]

## 8.1 Introduction

Until as recently as three years ago, reasoning was considered to be one of the most significant challenges that large language models (LLMs) had not yet overcome [Marcus, 2018, 2020, Garcez and Lamb, 2020]. Recently, LLMs have shown impressive success on a wide range of reasoning tasks, including commonsense [Wei et al., 2021, Sanh et al., 2021, Madaan et al., 2022c], mathematical [Lewkowycz et al., 2022, Wu et al., 2022a, Mishra et al., 2022a], and symbolic reasoning [Yao et al., 2022, Ahn et al., 2022], using few-shot prompting [Brown et al., 2020b].

This process has been accelerated by methods that require LLMs to generate their explicit reasoning steps, such as "chain-of-thought" [Wei et al., 2022c], "scratchpads" [Nye et al., 2021a],

[1] Code and data at http://reasonwithpal.com.

and "least-to-most" [Zhou et al., 2022] prompting. In particular, the widely used chain-of-thought (CoT) method presents the model with the explicit intermediate steps that are required to reach the final answer. Then, the model is expected to apply a similar decomposition to the actual test example, and consecutively reach an accurate final answer [Ling et al., 2017, Amini et al., 2019]. Nevertheless, while LLMs can decompose natural language problems into steps and perform *simple* arithmetic operations, their performance falls dramatically when dealing with complex arithmetic [Hendrycks et al., 2021, Madaan and Yazdanbakhsh, 2022a] or large numbers [Geva et al., 2020, Nogueira et al., 2021, Qian et al., 2022]. In fact, even when fine-tuning a PaLM-based model on 164B tokens of explicit mathematical content, its two most common failures are reportedly "incorrect reasoning" and "incorrect calculation" [Lewkowycz et al., 2022].

In this paper, we propose Program-Aided Language model (PAL): a novel approach that uses an LLM to read natural language problems and generate *programs* as reasoning steps, but offloads the *solution* step to a Python interpreter, as illustrated in Figure 8.1. This offloading leverages an LLM that can decompose a natural language problem into programmatic steps, which is fortunately available using contemporary state-of-the-art LLMs that are pre-trained on both natural language and programming languages [Brown et al., 2020b, Chen et al., 2021c, Chowdhery et al., 2022a]. While natural language understanding and decomposition *require* LLMs, solving and reasoning can be done with the external solver. This bridges an important gap in chain-of-thought-like methods, where reasoning chains can be correct but produce an incorrect answer.

We demonstrate the effectiveness of PAL across **13** arithmetic and symbolic reasoning tasks. In all these tasks, PAL using Codex [Chen et al., 2021c] outperforms much larger models such as PaLM-540B using chain-of-thought prompting. For example, on the popular Math Reasoning benchmark, PAL achieves state-of-the-art accuracy, surpassing PaLM-540B with chain-of-thought by absolute 15% top-1 accuracy. When the questions contain large numbers, a dataset we call GSM-HARD, PAL outperforms CoT by an absolute 40%. We believe that this seamless synergy between a neural LLM and a symbolic interpreter is an essential step towards general and robust AI reasoners.

## 8.2 Background: Few-shot Prompting

Few-shot prompting leverages the strength of large-language models to solve a task with a set of $k$ examples that are provided as part of the test-time input [Brown et al., 2020b, Liu et al., 2021b, Chowdhery et al., 2022a], where $k$ is usually a number in the low single digits. These input-output examples $\{(x_i, y_i)\}_{i=1}^{k}$ are concatenated in a prompt p $\equiv \langle x_1 \cdot y_1 \rangle \parallel \langle x_2 \cdot y_2 \rangle \parallel \ldots \parallel \langle x_k \cdot y_k \rangle$. where "$\cdot$" denotes the concatenation of an input and output, and "$\parallel$" indicate the concatenation of different examples. During inference, a test instance $x_{test}$ is appended to the prompt, and $p \parallel x_{test}$ is passed to the model which attempts to complete $p \parallel x_{test}$, and thereby generate an answer $y_{test}$. Note that such few-shot prompting does not modify the underlying LLM.

Wei et al. [2022c] additionally augment each in-context example with *chain of thought* (CoT) intermediate steps. Specifically, each in-context example in the CoT setup is a triplet $\langle x_i, t_i, y_i \rangle$, where $x_i$ and $y_i$ are input-output pair as before, and $t_i$ is a natural language description of the steps that are needed to arrive at the output $y_i$ from the input $x_i$. See Figure 8.1 for an example. With

the additional "thoughts" $t_i$, the prompt is set to $\text{p} \equiv \langle x_1 \cdot t_1 \cdot y_1 \rangle \parallel \langle x_2 \cdot t_2 \cdot y_2 \rangle \parallel \ldots \parallel \langle x_k \cdot t_k \cdot y_k \rangle$.

During inference, the new question $x_{test}$ is appended to the prompt as before and supplied to the LLM. Crucially, the model is tasked with generating *both* the thought $t_{test}$ and the final answer $y_{test}$. This approach of prompting the model to first generate a reasoning process $t_{test}$ improves the accuracy of the answer $y_{test}$ across a wide range of tasks [Wang et al., 2022a, Wei et al., 2022c, Zhou et al., 2022, Wang et al., 2022b].

## 8.3 Program-aided Language Models

In a Program-aided Language model, we propose to generate the thoughts $t$ for a given natural language problem $x$ as interleaved natural language (NL) and programming language (PL) statements. Since we delegate the solution step to an interpreter, we do not provide the final answers to the examples in our prompt. That is, every in-context example in PAL is a *pair* $\langle x_i, t_i \rangle$, where $t_j = [s_1, s_2, \ldots, s_N]$ with each $s_i \in \text{NL} \cup \text{PL}$, a sequence of tokens in either NL or PL. The complete prompt is thus $\text{p} \equiv \langle x_1 \cdot t_1 \rangle \parallel \langle x_2 \cdot t_2 \rangle \parallel \ldots \parallel \langle x_k \cdot t_k \rangle$.

Given a test instance $x_{test}$, we append it to the prompt, and $p \parallel x_{test}$ is fed to the LM. We let the LM generate a prediction $t_{test}$, which contains both the intermediate steps *and* their corresponding programmatic statements.

**Example** A close-up of the example from Figure 8.1 is shown in Figure 8.2.

While chain-of-thought only decomposes the solution in the prompt into natural language steps such as Roger started with 5 tennis balls and 2 cans of 3 tennis balls each is 6, in PAL we also augment each such NL step with its corresponding programmatic statement such as `tennis_balls = 5` and `bought_balls = 2 * 3`. This way, the model learns to generate a *program* that will provide the answer for the test question, instead of relying on LLM to perform the calculation correctly.

We prompt the language model to generate NL intermediate steps using comment syntax (e.g. "# ..." in Python) such they will be ignored by the interpreter. We pass the generated program $t_{test}$ to its corresponding solver, we run it, and obtain the final run result $y_{test}$. In this work we use a standard Python interpreter, but this can be any solver, interpreter or a compiler.

**Crafting prompts for PAL** In our experiments, we leveraged the prompts of existing work whenever available, and otherwise randomly selected the same number (3-6) of examples as previous work for creating a fixed prompt for every benchmark. In all cases, we augmented the free-form text prompts into PAL-styled prompts, leveraging programming constructs such as `for` loops and dictionaries when needed. Generally, writing PAL prompts is easy, and does not require more effort than writing the initial COT prompts.

We also ensure that variable names in the prompt meaningfully reflect their roles. For example, a variable that describes the *number of apples in the basket* should have a name such as `num_apples_in_basket`. This keeps the generated code linked to the entities in the question. In Section 8.6, we show that such meaningful variable names are critical to the downstream performance. Notably, it is also possible to incrementally run the PL segments and feed the

| | GSM | GSM-HARD | SVAMP | ASDIV | SINGLEEQ | SINGLEOP | ADDSUB | MULTIARITH |
|---|---|---|---|---|---|---|---|---|
| DIRECT $_{\text{Codex}}$ | 19.7 | 5.0 | 69.9 | 74.0 | 86.8 | 93.1 | 90.9 | 44.0 |
| COT $_{\text{UL2-20B}}$ | 4.1 | - | 12.6 | 16.9 | - | - | 18.2 | 10.7 |
| COT $_{\text{LaMDA-137B}}$ | 17.1 | - | 39.9 | 49.0 | - | - | 52.9 | 51.8 |
| COT $_{\text{Codex}}$ | 65.6 | 23.1 | 74.8 | 76.9 | 89.1 | 91.9 | 86.0 | 95.9 |
| COT $_{\text{PaLM-540B}}$ | 56.9 | - | 79.0 | 73.9 | 92.3 | 94.1 | 91.9 | 94.7 |
| COT $_{\text{Minerva 540B}}$ | 58.8 | - | - | - | - | - | - | - |
| PAL | **72.0** | **61.2** | **79.4** | **79.6** | **96.1** | **94.6** | **92.5** | **99.2** |

Table 8.1: Problem solve rate (%) on mathematical reasoning datasets. The highest number on each task is in **bold**. The results for DIRECT and PaLM-540B are from Wei et al. [2022c], the results for LaMDA and UL2 are from Wang et al. [2022b], and the results for Minerva are from Lewkowycz et al. [2022]. We ran PAL on each benchmark 3 times and report the average; the standard deviation is provided in Table 53.

execution results back to the LLM to generate the following blocks. For simplicity, in our experiments, we used a single, post-hoc, execution.

## 8.4 Experimental Setup

**Data and in-context examples** We experiment with three broad classes of reasoning tasks: (1) mathematical problems (§8.4.1) from a wide range of datasets including Math Reasoning [Cobbe et al., 2021], SVAMP [Patel et al., 2021], ASDIV [Miao et al., 2020], and MAWPS [Koncel-Kedziorski et al., 2016]; (2) symbolic reasoning (§8.4.2) from BIG-Bench Hard [Suzgun et al., 2022a]; (3) and algorithmic problems (§8.4.3) from BIG-Bench Hard as well. Details of all datasets are shown in subsection H.8. For all of the experiments for which COT prompts were available, we used the same in-context examples as used by previous work. Otherwise, we randomly sampled a fixed set of in-context examples, and used the same set for PAL and COT.

**Baselines** We consider three prompting strategies: DIRECT prompting – the standard prompting approach using pairs of questions and immediate answers (e.g., Answer: 11 ) as in Brown et al. [2020b]; chain-of-thought (COT) prompting [Wei et al., 2022c]; and our PAL prompting. We performed greedy decoding from the language model using a temperature of 0. Unless stated otherwise, we used CODEX (`code-davinci-002`) as our backend LLM for both PAL, DIRECT, and COT. In datasets where results for additional base LMs, such as PaLM-540B, were available from previous work, we included them as COT $_{\text{PaLM-540B}}$.

### 8.4.1 Mathematical Reasoning

We evaluate PAL on eight mathematical word problem datasets. Each question in these tasks is an algebra word problem at grade-school level. An example for a question and PAL example prompt is shown in Figure 8.3. We found that using explicit NL intermediate steps does not further benefit these math reasoning tasks, hence we kept only the meaningful variable names in the prompt.

| | COLORED OBJECT | PENGUINS | DATE | REPEAT COPY | OBJECT COUNTING |
|---|---|---|---|---|---|
| DIRECT $_{\text{Codex}}$ | 75.7 | 71.1 | 49.9 | 81.3 | 37.6 |
| COT $_{\text{LaMDA-137B}}$ | - | - | 26.8 | - | - |
| COT $_{\text{PaLM-540B}}$ | - | 65.1 | 65.3 | - | - |
| COT $_{\text{Codex}}$ | 86.3 | 79.2 | 64.8 | 68.8 | 73.0 |
| PAL $_{\text{Codex}}$ | **95.1** | **93.3** | **76.2** | **90.6** | **96.7** |

Table 8.2: Solve rate on three symbolic reasoning datasets and two algorithmic datasets, In all datasets, PAL achieves a much higher accuracy than chain-of-thought. Results with closed models LaMDA-137B and PaLM-540B are included if available to public Wei et al. [2022c], Suzgun et al. [2022a].

### 8.4.2 Symbolic Reasoning

We applied PAL to three symbolic reasoning tasks from BIG-Bench Hard [Suzgun et al., 2022a], which involve reasoning about objects and concepts: (1) COLORED OBJECTS requires answering questions about colored objects on a surface. This task requires keeping track of relative positions, absolute positions, and the color of each object. Figure 8.4 shows an example for a question and example PAL prompt. (2) PENGUINS describes a table of penguins and some additional information in natural language, and the task is to answer a question about the attributes of the penguins, for example, "*how many penguins are less than 8 years old?*". While both PENGUINS and COLORED OBJECT tasks require tracking objects, PENGUINS describes *dynamics* as well, since the penguins in the problem can be added or removed. Figure 50 in Appendix H.11 shows an example for a question, a chain-of-thought prompt, and PAL prompt. (3) DATE is a date understanding task that involves inferring dates from natural language descriptions, performing addition and subtraction of relative periods of time, and having some global knowledge such as "how many days are there in February", and performing the computation accordingly. Appendix H.11 shows example prompts.

### 8.4.3 Algorithmic Tasks

Finally, we compare PAL and COT on algorithmic reasoning. These are tasks where a human programmer can write a deterministic program with prior knowledge of the question. We experiment with two algorithmic tasks: OBJECT COUNTING and REPEAT COPY. OBJECT COUNTING involves answering questions about the number of objects belonging to a certain type. For example, as shown in Figure 8.5: *"I have a chair,* ***two potatoes****,* ***a cauliflower****,* ***a lettuce head****, two tables, ... How many vegetables do I have?"*). REPEAT COPY requires generating a sequence of words according to instructions. For example, as shown in Appendix H.11: *"Repeat the word duck four times, but halfway through also say quack"*).

## 8.5 Results

### 8.5.1 Math Results

Table 8.1 shows the following results: across all tasks, PAL using Codex sets a new few-shot state-of-the-art top-1 decoding across all datasets, outperforming $\text{COT}_{\text{Codex}}$, $\text{COT}_{\text{PaLM-540B}}$, and $\text{COT}_{\text{Minerva 540B}}$ which was fine-tuned on explicit mathematical content.

Interestingly, COT also benefits from Codex over PaLM-540B in some of the datasets such as ASDIV, but performs worse than PaLM-540B in others such as SVAMP. Yet, using PAL further improves the solve rate across all datasets.

**GSM-HARD** LLMs can perform simple calculations with *small* numbers. However, Madaan and Yazdanbakhsh [2022a] found that 50% of the numbers in the popular Math Reasoning dataset of math reasoning problems are *integers between 0 and 8*. This raises the question of whether LLMs can generalize to larger and non-integer numbers? We constructed a harder version of Math Reasoning, which we call GSM-HARD, by replacing the numbers in the questions of Math Reasoning with larger numbers. Specifically, one of the numbers in a question was replaced with a random integer of up to 7 digits. More details regarding the this new dataset are provided in H.9. On GSM-HARD (Table 8.1), the accuracy of DIRECT drops dramatically from 19.7% to 5.0% (a relative drop of 74%), the accuracy of COT drops from 65.6% to 20.1% (a relative drop of almost 70%), while PAL remains stable at 61.5%, dropping by only 14.3%. The results of COT on GSM-HARD did not improve even when we replaced *its prompts* with prompts that include large numbers (Appendix H.2).

This shows how PAL provides not only better results on the standard benchmarks, but is also much more *robust*. In fact, since PAL offloads the computation to the Python interpreter, any complex computation can be performed accurately given the correctly generated program.

**Large Numbers or Incorrect Reasoning?** Are the failures on GSM-HARD primarily due to the inability of LLMs to do arithmetic, or do the large numbers in the question "confuse" the LM which generates irrational intermediate steps? To investigate this, we evaluated the outputs generated by COT for the two versions of the same question (with and without large numbers). We find that in 16 out of 25 cases we analyzed, COT generates nearly identical natural language "thoughts", indicating that the primary failure mode is the inability to perform arithmetic accurately. Sample outputs are provided in the Appendix, Table 57.

**Multi-sample Generation** As found by Wang et al. [2022b], chain-of-thought-style methods can be further improved by sampling $k > 1$ outputs, and selecting the final answer using majority voting. We thus repeated the greedy-decoding experiments using nucleus sampling [Holtzman et al., 2020] with $p = 0.95$ and $k = 40$ as in Lewkowycz et al. [2022] and temperature of 0.7. As shown in Table 8.3, this further increases the accuracy of PAL from 72.0% to 80.4% on Math Reasoning, obtaining 1.9% higher accuracy than Minerva-540B using the same number of samples.

| | Math Reasoning `majority@40` |
|---|---|
| CoT $_{\text{UL2-20B}}$ | 7.3 |
| CoT $_{\text{LaMDA-137B}}$ | 27.7 |
| CoT $_{\text{Codex}}$ | 78.0 |
| CoT $_{\text{PaLM-540B}}$ | 74.4 |
| CoT $_{\text{Minerva 540B}}$ | 78.5 |
| PAL $_{\text{Codex}}$ | **80.4** |

Table 8.3: Problem solve rate (%) on Math Reasoning using `majority@40` [Wang et al., 2022b]

### 8.5.2 Symbolic Reasoning & Algorithmic Tasks Results

Results for symbolic reasoning and algorithmic tasks are shown in Table 8.2. In COLORED OBJECTS, PAL improves over the strong COT by 8.8%, and by 19.4% over the standard direct prompting. In PENGUINS, PAL provides a gain of absolute 14.1% over COT. In DATE, PAL further provides 11.4% gain over both COT $_{\text{Codex}}$, $_{\text{PaLM-540B}}$, and $_{\text{LaMDA-137B}}$.

The two rightmost columns of Table 8.2 show that PAL is close to solving OBJECT COUNTING, reaching 96.7% and improving over COT by absolute 23.7%. Similarly, PAL vastly outperforms COT by absolute 21.8% on REPEAT COPY. Surprisingly, DIRECT prompting performs better than COT on REPEAT COPY. Yet, PAL improves over DIRECT by 9.3% in REPEAT COPY.

**Is PAL sensitive to the complexity of the question?** We examined how the performance of PAL and COT changes as the complexity of the input question grows, measured as the number of objects in the question of COLORED OBJECTS. As shown in Figure 8.6, PAL outperforms COT across all input lengths. As the number of objects in the question increases, COT's accuracy is unstable and drops, while PAL remains consistently close to 100%. More analysis on the token-level predictions can be found in Appendix H.7.

## 8.6 Analysis

**Does PAL work with weaker LMs?** In all our experiments in Section 8.5, PAL used the `code-davinci-002` model; but can PAL work with weaker models of code? We compared PAL with COT when both prompting approaches use the same weaker base LMs `code-cushman-001` and `code-davinci-001`. As shown in Figure 8.7, even though the absolute accuracies of `code-cushman-001` and `code-davinci-001` are lower, the relative improvement of PAL over COT remains consistent across models. This shows that PAL can work with weaker models, while its benefit scales elegantly to stronger models as well.

**Is PAL limited to Code-LMs?** We also experimented with PAL using the *`text`*`-davinci` series. Figure 8.8 shows the following interesting results: when the base LM's "code modeling ability" is weak (using `text-davinci-001`), COT performs better than PAL. However, once the LM's code modeling ability is sufficiently high (using `text-davinci-002` and

`text-davinci-003`), PAL outperforms COT, and PAL $_{\texttt{text-davinci-003}}$ performs almost as PAL $_{\texttt{code-davinci-002}}$. This shows that PAL is not limited to LMs of code, but it can work with LMs that were mainly trained for natural language, if they have a sufficiently high coding ability.

The base *ChatGPT* (`gpt-3.5-turbo`) appears to be stronger than the base `text-davinci-003` on Math Reasoning. However as shown in Figure 8.8, PAL *further improves* even the strong ChatGPT model by 2.8% absolute.

**Is PAL better because of the Python prompt or because of the interpreter?** We experimented with generating Python code, while requiring the neural LM to "execute" it as well, without using an interpreter, following Nye et al. [2021a], Madaan et al. [2022c]. We created prompts that are similar to PAL's, except that they *do include* the final answer. This resulted in a 23.2 solve rate on Math Reasoning, much lower than PAL (72.0), and only 4.5 points higher than DIRECT. These results reinforce our hypothesis that the main benefit of PAL comes from the synergy with the interpreter, and not only from having a better prompt. Additional details are provided in Appendix H.2. For additional discussion on code-prompts compared to textual-prompts, see Appendix H.7.

**Do variable names matter?** In all our experiments, we used meaningful variable names in the PAL prompts, to ease the model's grounding of variables to the entities they represent. For the Python interpreter, however, variable names are meaningless. To measure the importance of meaningful variable names, we experimented with two prompts variants:

1. PAL$_{-\text{comment}}$ – the PAL prompt without intermediate NL comments.
2. PAL$^{-\text{var}}_{-\text{comment}}$ – the PAL prompt without intermediate NL comments *and* with variable names substituted with random characters.

The results are shown in Figure 8.9. In COLORED OBJECTED and DATE, removing intermediate NL comments but keeping meaningful variable names (PAL$_{-\text{comment}}$) – slightly reduces the accuracy compared to the full PAL prompt, but it still achieves higher accuracy than the baselines COT. Removing variable names as well (PAL$^{-\text{var}}_{-\text{comment}}$) further decreases accuracy, and performs worse than COT. Since variable names have an important part in code quality [Gellenbeck and Cook, 1991, Takang et al., 1996], meaningful variable names are only expected to ease reasoning for Codex, which was trained on mostly meaningful names, as was also found by Madaan et al. [2022c].

**Is the generated code correct?** *All* generated Python code was syntactically correct. Less than 1% of the examples raised an exception at runtime; among these examples, the most common error was trying to use a variable that was not defined before (`NameError`).

## 8.7 Related Work

**Prompting** Few-shot prompting Brown et al. [2020b] has been shown to be an effective approach for a variety of tasks [Liu et al., 2021b] ranging from text- [Gehrmann et al., 2021a, Reif et al., 2021, Wei et al., 2021, Sanh et al., 2021] to code-generation [Chen et al., 2021b]. Methods such

as chain-of-thought prompting (CoT) have further unlocked a variety of reasoning tasks, boosting the performance of models on a variety of benchmarks. Nevertheless, all previous approaches suffer from inaccuracy in arithmetic calculation and incorrect reasoning [Lewkowycz et al., 2022, Hendrycks et al., 2021, Madaan and Yazdanbakhsh, 2022a]. PAL avoids these problems by offloading the calculation and some of the reasoning to a Python interpreter, which is correct by construction, given the right program.

**LMs with external tools** Several prior works have equipped neural models with specialized modules to create effective cascades [Dohan et al., 2022]. For example, Cobbe et al. [2021] employ a calculator for arithmetic operations as a post hoc processing, and Demeter and Downey [2020] add specialized modules for generating cities and dates. Unlike these works, PAL generates code for a Python interpreter, which is general enough to handle both arithmetic calculations and dates, without specialized modules and ad-hoc fixes. Chowdhery et al. [2022a] and Wei et al. [2022c] have also experimented with external calculators; however, the calculator had improved Codex by only 2.3% (absolute) on Math Reasoning and improved PaLM-540B by 1.7%, while PAL improves Codex by 6.4% on the same benchmark (Section 8.5.1). Similarly to our work, Chowdhery et al. [2022a] have also experimented with generating Python code for solving the Math Reasoning benchmark, but their experiments resulted in *lower* accuracy than the standard PaLM-540B that uses chain-of-thought. Pi et al. [2022] pretrain the model on execution results of random expressions on a calculator, instead of using the solver at test time as well. While their model can hypothetically perform arithmetic better than other pretrained LMs, their results on the SVAMP benchmark are much lower: 57.4% using a T5-11B model, while PAL achieves 79.4% on the same benchmark without any specialized pretraining.

Shortly after a preprint of our work was submitted to arXiv, another related work on "program of thought prompting" Chen et al. [2022] was also submitted to arXiv. Their method is conceptually similar to ours, but Chen et al. [2022] (1) only demonstrates efficacy on mathematical problems, whereas we demonstrate gains on symbolic and algorithmic benchmarks as well, and (2) chose benchmark-specific prompt examples, while we used the same prompt examples as previous work, to disentangled the benefit of our approach from the benefit of the choice of examples.

**Semantic parsing** Our work can also be seen as a very general form of semantic parsing, where instead of parsing into strict domain-specific languages, the model generates free-form Python code. Some works constrain the decoder using a Context-Free Grammar (CFG) to generate a domain-specific meaning representation [Shin and Van Durme, 2022] or a canonical utterance, which can be converted to a Lisp-like meaning representation [Shin et al., 2021]. Nye et al. [2021b] first generate candidate sentences using a language model; then, another language model (GPT-3) is used to derive logical constraints entailed by each candidate; these constraints are then cross-checked with a predefined list of facts to rank the candidates. In contrast, PAL does not require any constraining or domain-specific representations other than Python code. Further, LMs that were pretrained on Python are abundant compared to other domain-specific languages, making Python code a much more preferable representation. Andor et al. [2019] generate task-specific arithmetic operations for reading comprehension tasks; Gupta et al. [2020] design neural modules

such as `count` to deal with arithmetic operations. PAL generalizes these works by generating general Python programs, without the need for defining specialized modules. The closest work to ours technically may be Binder [Cheng et al., 2022], but it addressed mostly answering questions about tables using SQL and SQL-like Python.

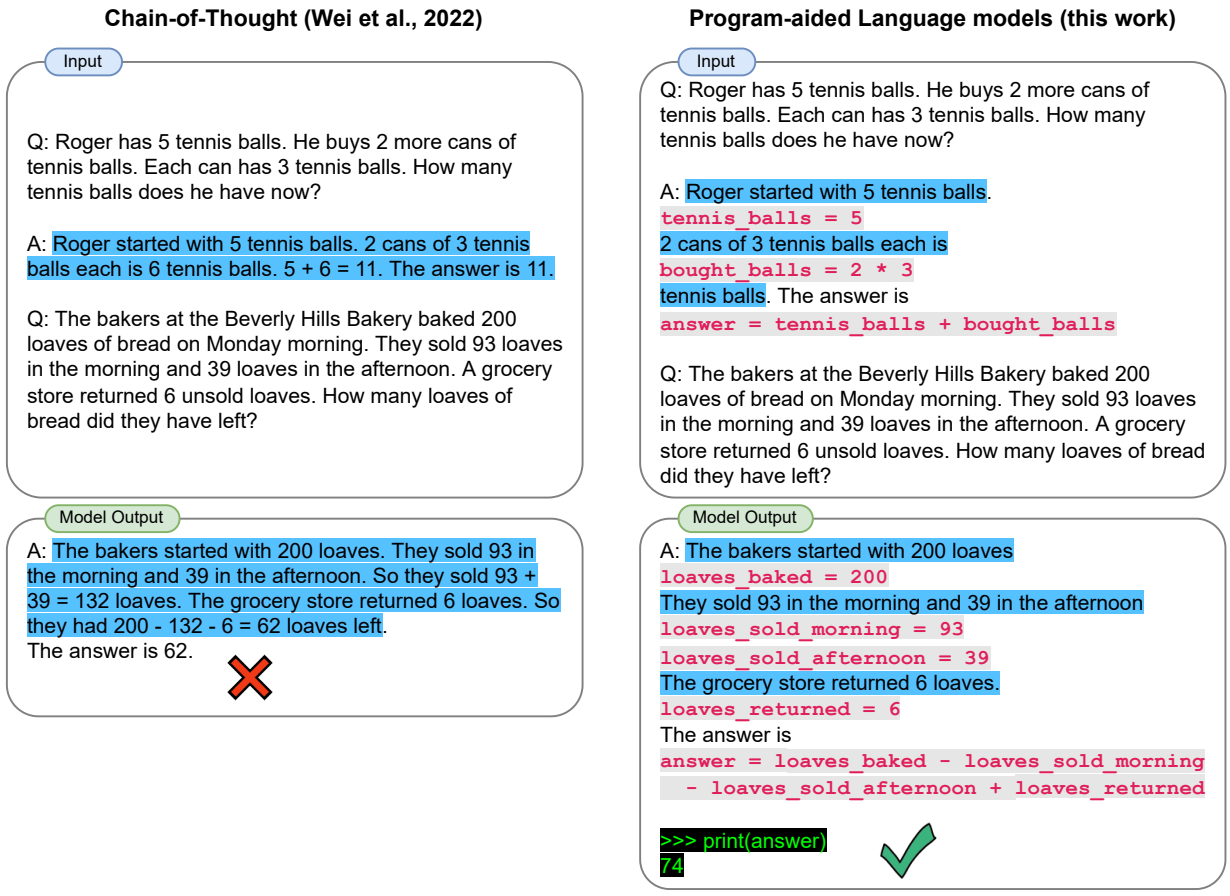



Figure 8.1: A diagram illustrating PAL: Given a mathematical reasoning question, Chain-of-thought (left) generates intermediate reasoning steps of free-form text. In contrast, Program-aided Language models (PAL, right) generate intermediate steps *and* Python code. This shifts the role of *running* the reasoning steps from the language model to the Python interpreter. The final answer is obtained by running the generated reasoning chain. Chain-of-thought reasoning is highlighted in blue; PAL steps are `highlighted in gray and pink`; the Python interpreter run is highlighted in black and green.

A: Roger started with 5 tennis balls.
`tennis_balls = 5`
2 cans of 3 tennis balls each is
`bought_balls = 2 * 3`
tennis balls. The answer is
`answer = tennis_balls + bought_balls`

Figure 8.2: A close-up of a single example from a PAL prompt. Chain-of-thought reasoning is highlighted in blue, and PAL programmatic steps are `highlighted in gray and pink`.

Q: Olivia has $23. She bought five bagels for $3 each. How much money does she have left?

```
money_initial = 23
bagels = 5
bagel_cost = 3
money_spent = bagels * bagel_cost
money_left = money_initial -
    money_spent
answer = money_left
```

Figure 8.3: Example prompt for the mathematical reasoning tasks, from the Math Reasoning benchmark.

Q: On the table, you see a bunch of objects arranged in a row: a purple paperclip, a pink stress ball, a brown keychain, a green scrunchiephone charger, a mauve fidget spinner, and a burgundy pen. What is the color of the object directly to the right of the stress ball?

```
...
stress_ball_idx = None
for i, object in enumerate(objects):
    if object[0] == 'stress ball':
        stress_ball_idx = i
        break
# Find the directly right object
direct_right =
    objects[stress_ball_idx+1]
# Check the directly right object's
    color
answer = direct_right[1]
```

Figure 8.4: An example for a PAL prompt in the COLORED OBJECTS task. For space considerations, we omit the code that creates the list `objects`.

Q: I have a chair, two potatoes, a cauliflower, a lettuce head, two tables, a cabbage, two onions, and three fridges. How many vegetables do I have?

```
# note: I'm not counting the chair,
    tables, or fridges
vegetables_to_count = {
    'potato': 2,
    'cauliflower': 1,
    'lettuce head': 1,
    'cabbage': 1,
    'onion': 2
}
answer =
    sum(vegetables_to_count.values())
```

Figure 8.5: An example for a PAL prompt in the OBJECT COUNTING task. The base LM is expected to convert the input into a dictionary where keys are entities and values are their quantities, while filtering out non-vegetable entities. Finally, the answer is the sum of the dictionary values.

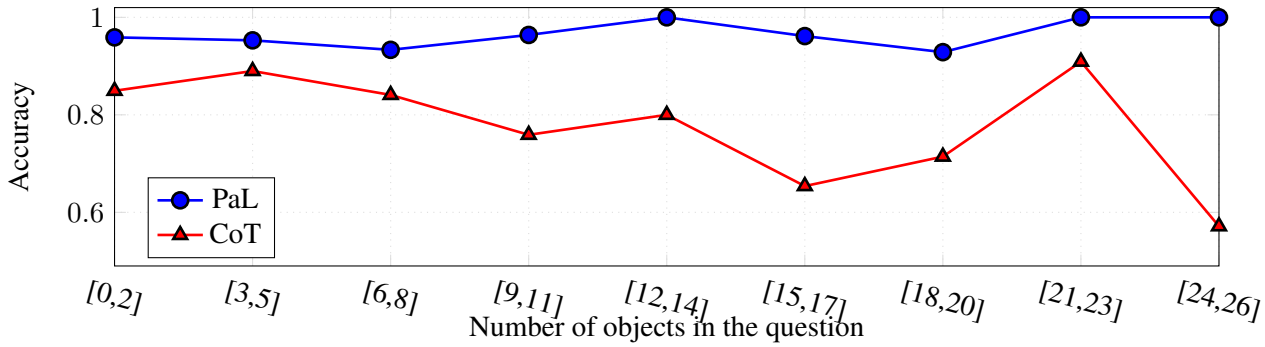



Figure 8.6: The solve rate on COLORED OBJECTS with respect to the number of objects included in the test question.

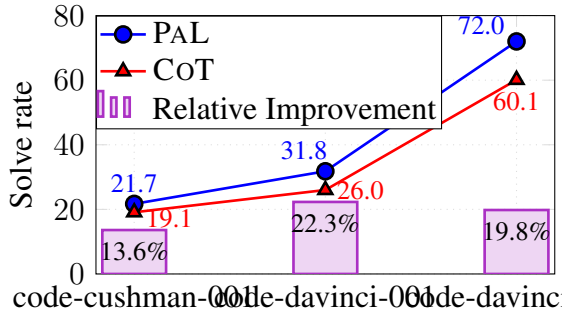



Figure 8.7: PAL with different models on Math Reasoning: though the absolute accuracies with `code-cushman-001` and `code-davinci-001` are lower than `code-davinci-002`, the relative improvement of PAL over COT is consistent across models.

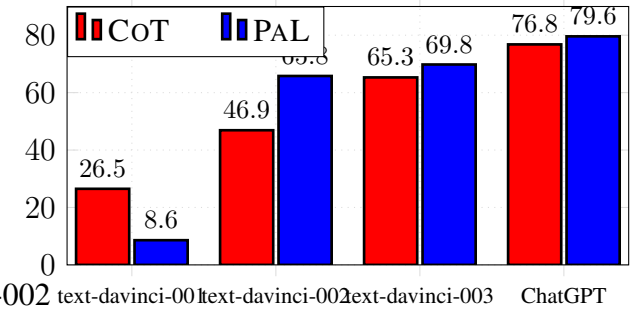



Figure 8.8: PAL with NL LMs on Math Reasoning: though COT outperforms PAL with `text-davinci-001`, once the base LM is sufficiently strong, PAL is beneficial with `text-davinci-002` and `text-davinci-003` as well. That is, PAL is not limited to code-LMs only.

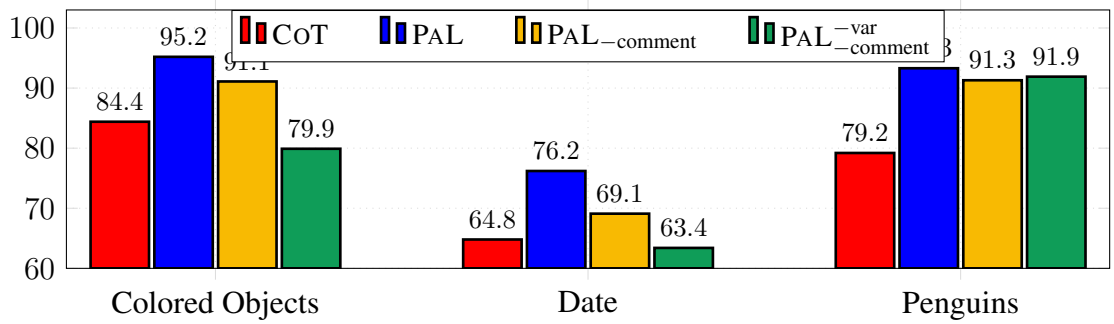



Figure 8.9: Ablation study of PAL prompt formats. We consider the original PAL prompt, it with natural language comments removed ($\text{PAL}_{-\text{comment}}$), and further variable names replaced with random character ($\text{PAL}^{-\text{var}}_{-\text{comment}}$). As a reference, we also show the COT performance (blue).

# Part IV

# Post-Inference Enchancements for LLMs

In this part of the thesis, we delve into the realm of post-inference enhancements for large language models (LLMs), showcasing how structure can be harnessed even after the primary stages of pre-training, fine-tuning, and inference. The chapters in this part demonstrate that there are still opportunities to infuse structure and improve LLM outputs through user interactions and iterative refinement, further emphasizing the significance of structure throughout the entire lifecycle of LLMs.

The growing capabilities of LLMs have led to remarkable progress in various natural language processing tasks. However, there remains room for improvement in their outputs, particularly in the context of user interactions and incorporating feedback. By focusing on post-inference enhancements, we aim to address these limitations and ensure that LLMs provide even more accurate and useful results.

In this part, we present two chapters that explore different approaches to enhancing LLMs post-inference:

1. Chapter 9 introduces MEMPROMPT, an approach that pairs GPT-3 with a memory of user feedback for improved accuracy across diverse tasks. By recording misunderstandings and user feedback, the system generates enhanced prompts for new queries based on past user interactions. A variant of MEMPROMPT, FB-NET, leverages feedback to fix mistakes in the outputs of a fine-tuned model for structured generation.
2. Chapter 10 presents Self-Refine, a framework for iteratively refining LLM outputs by generating multi-aspect feedback. This approach does not require supervised training data or reinforcement learning and works with a single LLM. Tested on various tasks, Self-Refine outperforms direct generation and shows improvements over outputs generated directly with GPT-3.5 and GPT-4. Despite its effectiveness, the current Self-Refine framework is limited in its expressiveness. The loop of generate output, get feedback, and refine output is currently linear. However, humans often create non-trivial outputs non-linearly. In this part, we propose to explore planning approaches for non-linear Self-Refine.

Through these chapters, we highlight the importance of post-inference enhancements and how they can further improve LLM outputs. This part of the thesis underscores the ongoing potential of structure in the development and application of large language models.

# Chapter 9

# MemPrompt: Memory-assisted Prompt Editing with User Feedback

## 9.1 Introduction

Language models are now better than ever before at generating realistic content, but still lack commonsense Bender and Koller [2020], Marcus [2021]. One failure mode due to a lack of commonsense is in misunderstanding a user's *intent*. The typical remedy of retraining with more data is prohibitive due to the cost and infrastructure requirements. In such cases, even if users repeatedly observe the model making a mistake, there are no avenues to provide feedback to the model to make it more accurate and personalized over time.

Our goal is to allow users to correct such errors directly through interaction, and without retraining by injecting the knowledge required to correct the model's misunderstanding. Building upon the recent success of injecting commonsense in the input [Lewis et al., 2020b, Talmor et al., 2020], we propose a novel approach of injecting knowledge in the input via interactive feedback from an end-user.

Our approach, MemPrompt, pairs GPT-3 with a growing memory of cases where the model misunderstood user's intent and was provided with corrective feedback. This feedback is question dependent, and thus the prompt for each sample is *edited* to adapt to the input. In this sense, our work can be seen as an instance of prompt engineering Liu et al. [2021a] which involves editing the prompts. Our work adds interactivity to prompt engineering as it involves dynamically updating the prompt for every instance.

Figure 9.2 presents a sample interaction between a user and GPT-3 that our setup enables. The model was asked for a similar word. However, the model's (incorrect) task understanding a was "The homophone of good is". The user can detect such discrepancy between the intended and interpreted task instruction, and can provide feedback $fb$ as "*similar to* means *with a similar meaning*", clarifying that they actually wanted a synonym. Crucially, note that such instructional correction is feasible *even if the user does not know the correct answer to their question*, as they are critiquing the model's understanding of their intent, rather than the answers themselves. Thus, our setup **does not** require the users to be experts at tasks being solved, another advantage of our approach.

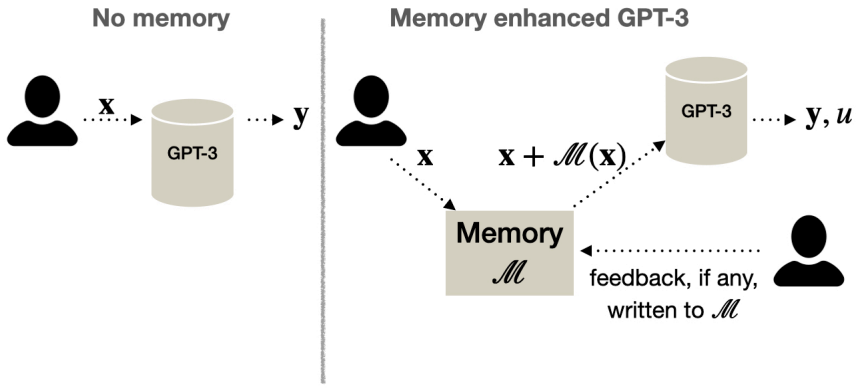



Figure 9.1: Proposed architecture: (left) GPT-3 does not account for user feedback. (right) MemPrompt maintains a memory $\mathcal{M}$ of corrective feedback, and searches for feedback from prior queries with a similar intent as $x$ using a retrieval function $\mathcal{M}(\mathbf{x})$. $x$ is then concatenated to the retrieved feedback and appended to the prompt for querying GPT-3. Users can also give new feedback on the model's task understanding $u$, then added to $\mathcal{M}$.

Further, it is desirable to have a system that can leverage past feedback on new, unseen examples for prompt-editing. We maintain a memory $\mathcal{M}$ of such feedback as a set of key-value pairs, where the key is a misunderstood question, and the value is the user's feedback to correct that misunderstanding. Given a new question, we check if the model has made a mistake on a similar question earlier, by querying the memory for a similar question. If found, append the corresponding feedback to the question prompt. This mechanism aims to prevent the model from making the same type of mistake twice. This failure-driven reminding mechanism draws inspiration from the theory of recursive reminding in psychology Jacoby and Wahlheim [2013], which suggests humans index error corrections in the context in which those errors occurred.

This paper presents the general architecture for the system and provides representative implementations for each component. We then demonstrate the system on four tasks, using simulated user feedback: (1) lexical relations (e.g., antonyms, Figure 9.2), (2) word scrambling (e.g., anagrams), (3) ethical reasoning with user feedback being the appropriate *class* of ethical consideration, e.g., "it is about cheating", using a small set of categories, and (4) ethics reasoning with user feedback being natural language. We find that in all cases, GPT-3's accuracy significantly increases with time, without retraining, as our approach enables it to use corrective feedback from earlier examples to avoid similar misunderstandings on future examples. In summary, our **contributions** are:

- We show that a large model like GPT-3 can be improved after deployment, without retraining, through a memory-assisted architecture.
- Our implementation, MemPrompt, is the first demonstration that this is possible - this is an important step forward for real use of LMs, and the paper sets out a general architecture that others can build on, a specific implementation, and detailed evaluation on multiple tasks.

Our memory enhanced GPT-3 implementation.

**u** What word is similar to *good*?

**GPT-3:** The homophone of good is: wood.

**u** "Similar to" means "with similar meaning".

**GPT-3:** Noted *[writes to memory]*

**u** What word is similar to *surprised*?

**GPT-3:** The synonym of surprised is: amazed.

*[Retrieves and adds to prompt '"Similar to" means "with similar meaning"'].*

Figure 9.2: This paper enhances GPT-3 performance by looking up questions with a similar intent that received any user feedback. Our approach is simple because only the question in the prompt needs to be updated with relevant feedback, and no retraining is necessary.

## 9.2 Approach

### 9.2.1 Memory enhanced GPT-3 architecture

In our setup, given an input $\mathbf{x}$, a model generates an output $\mathbf{y}$ and a sentence a expressing its understanding of the task, a skill learned through few-shot examples in the prompt (Appendix I.4). The user can then critique a by providing natural language feedback $\mathbf{fb}$. This is feasible even if the user does not know the correctness of $\mathbf{y}$ because they are critiquing the *model's understanding of their intent* rather the answers themselves.

| Task ($\mathbf{fb}$ type) | ($\mathbf{x} \rightarrow \mathbf{y}$) | a and $\mathbf{fb}$ |
|---|---|---|
| Lexical relations (INS) | $\mathbf{x}$: What sounds like good?<br><br>$\mathbf{y}$: wood | a: Question is asking for a synonym.<br>$\mathbf{fb}$: No, I want a homophone. |
| Word scrambling (INS) | $\mathbf{x}$: Find the right word given this cycled word: elylarg<br>$\mathbf{y}$: largely | a: The question is about anagram.<br>$\mathbf{fb}$: No, its about uncycling a word. |
| Ethical reasoning (CAT) | $\mathbf{x}$: Turning my blender on at 3AM<br><br>$\mathbf{y}$: It's bad. | a: Question is about authority.<br><br>$\mathbf{fb}$: No, it is about harm. |
| Ethical reasoning (NL) | $\mathbf{x}$: John has started using again after his mother passed<br>$\mathbf{y}$: It's bad. | a: Question is about spending money.<br>$\mathbf{fb}$: No, it is about drug use. |

Table 9.1: Feedback types and demonstration of understanding: our system leverages user feedback to prevent failures caused due to a misunderstanding of the task.

Given a new query, MemPrompt uses $\mathbf{fb}$ from similar, prior queries to enrich the (few-shot) prompt $p$. We use the principle that if two inputs $x_i$ and $x_j$ are similar (i.e., $x_i \sim x_j$), then

their feedback $\mathbf{fb}_i$ and $\mathbf{fb}_j$ should be exchangeable ($x_i \sim x_j \Leftrightarrow fb_i \sim fb_j$). The underlying assumption here is that for a fixed model, similar inputs will incur similar errors, and thus can use the same feedback for correction. Fig. 9.1 gives an overview of MemPrompt, with the following components:

**Memory** $\mathcal{M}$ : $\mathcal{M}$ is a growing table of key ($\mathbf{x}_i$) - value ($\mathbf{fb}_i$) pairs that supports read, write, and lookup operations. The write operation is used whenever a user gives new feedback.

**Lookup** $\mathcal{M}(\mathbf{x})$ : The memory allows lookup operations, denoted as $\mathcal{M}(\mathbf{x})$, that matches the query=$\mathbf{x}$ against all the keys of $\mathcal{M}$.

**Combiner** $\mathcal{C}(\mathbf{x}, \mathcal{M}(\mathbf{x}))$ : A gating function allowing irrelevant, retrieved feedback to be ignored.

**Few-shot prompting** Let us briefly recap few-shot prompting with GPT-3. Consider a general setup where given an input $\mathbf{x}$, a model is expected to generate an output $\mathbf{y}$. In a few-shot prompting mode [Brown et al., 2020a], a prompt $p$ consists of $k$ $(\mathbf{x}, \mathbf{y})$ "in-context" examples, i.e., $p = \mathbf{x}_1.\mathbf{y}_1\#\mathbf{x}_2.\mathbf{y}_2 \ldots \#\mathbf{x}_k.\mathbf{y}_k$, where $\#$ is a token separating examples and . indicates concatenation. During inference, the user inputs a question $\mathbf{x}_i$, and the model is fed $p \# \mathbf{x}_i$ (i.e., the question suffixed to the prompt) and is expected to generate the answer $\mathbf{y}_i$ as a continuation.

**MemPrompt setup** As mentioned, given an input $\mathbf{x}$, we prompt the model to generate an output $\mathbf{y}$ and a sentence a expressing its understanding of the task. Thus, the in-context examples for MemPrompt are of the form $\mathbf{x} \rightarrow \text{a}, \mathbf{y}$. In addition to the input $\mathbf{x}$, MemPrompt retrieves a $\mathbf{fb}$ if a question similar to $\mathbf{x}$ has been asked before. To enable the model to react to such feedback, we also include examples of the form $(\mathbf{x}, fb \rightarrow \text{a}, \mathbf{y})$ in the prompt, which are aimed to teach the model to react to $fb$ (Appendix I.4).

### 9.2.2 Verbalizing Task Understanding

Existing methods for receiving user feedback typically assume the user knows the correct answer $\mathbf{y}$ Elgohary et al. [2021a]. This assumption is paradoxical: if the user knew the answer, why would they be using the model? Further, allowing only "oracle" users (who know correct $\mathbf{y}$) might lead to sampling biases. In real-world settings, it is common for users to not have the exact answer, but rather, a general understanding of what they are searching for. Thus, we propose eliciting a verbalization of task understanding a from the model in addition to the answer. End users can thus critique a.

We operationalize this idea by including task verbalization in the prompt (Fig. 9.3). Given a question *What sounds like ¡ sighted ¿ ?*, a vanilla prompting approach will generate the answer *cited*. In contrast, we include a a *the homophone for* in the prompt. Large-scale language models, such as GPT-3, have been shown to excel at reasoning with a limited number of examples, making them well-suited to mimic the prompt and generate not only the answer, but also an understanding of the task at hand. Given a test question *What sounds similar to ¡ sighted ¿ ?*, if the model generates *the word that has the same meaning* as a, the user has a reason to believe that the answer

is wrong. Our experiments demonstrate that GPT-3 models are able to generate this additional information in all tasks presented.

Our approach is not foolproof— the model may spell out a wrong a while giving out the correct answer, misleading the user into believing that there is an error (or vice-versa). Hallucinating remains a critical limitation of generative models Cao et al. [2022], therefore additional heuristics and model calibration might be necessary to make our approach foolproof. In practice, however, we found such cases to be rare for the tasks in this paper.

**(Word reasoning task)**
**Ques:** What sounds like ¡ sighted ¿ ?
**Ans:** the homophone for sighted is cited.

---

**(Ethical judgment task)**
**Situation:** i heard that if you cringe about your past it just means you've grown as a person, for anyone who needs to hear it.
**Morality Judgment:** This question is about: regretting poor decisions from your past. The answer is it's okay.

Figure 9.3: MemPrompt is tuned to generate task understanding + answer. This allows the users to provide feedback on the task understanding even without knowing the actual answer.

### 9.2.3 Allowing GPT-3 to react to feedback

Once the feedback is received from the user, can the model successfully utilize it? By adding a few examples of the form $\mathbf{x}, fb \rightarrow \mathrm{a}, \mathbf{y}$ in the prompt and setting $fb = \mathrm{a}$, we force the model to use the task understanding present in the input when generating the output (Figure 9.4). Recently, it has been shown that such repetition plays a crucial role in the success of few-shot prompting models [Madaan and Yazdanbakhsh, 2022b].

**Ques:** What is similar to popular ? clarification: when I ask for similar to, I want a synonym.
**Ans:** the synonym of popular is admired.

Figure 9.4: An in-context example of the form $\mathbf{x}, fb \rightarrow \mathrm{a}, \mathbf{y}$, which encourages a to be like $\mathbf{fb}$, thereby conditioning the output to react to $\mathbf{fb}$.

### 9.2.4 Feedback on model's understanding

Within the setup $\mathbf{x} \rightarrow \mathrm{a}, \mathbf{y}$, we focus on following two modes of failure:

- Task instruction understanding: this is especially concerning in a multi-tasking setup, where the model may consider the question to be about a different task than the one user intended.
- Task nuanced understanding: when the model understands the task type, but misunderstands the subtle intent in a question.

Our primary goal is to elicit feedback on the model's understanding of the task, however, we also explore settings where an Oracle is available to provide feedback on the labels (as detailed in Section I.6). Finally, we note again that the model reacts to the feedback because some in-context samples are of the form: $(\mathbf{x}, fb \rightarrow \text{a}, \mathbf{y})$. We consider a diverse set of tasks ($\mathbf{x} \rightarrow \mathbf{y}$), $\mathbf{fb}$ and a, as summarized in Table 9.1.

### 9.2.5 Tasks

We apply our approach to four tasks: (1) lexical relations (e.g., antonyms, Figure 9.2), (2) word scrambling (e.g., anagrams), (3) ethics (with user feedback being the appropriate *class* of ethical consideration), and (4) ethics (with user feedback being natural language). For all five tasks, the dataset consists of $(\mathbf{x}, fb \rightarrow \text{a}, \mathbf{y})$ tuples, where $\mathbf{fb}$ clarifies the task in $\mathbf{x}$. We have a simulated conversational setting, in which a user can ask the model $\mathbf{x}$ (covering any of these five tasks). If the model gives a wrong answer to query $\mathbf{x}$, then $\mathbf{fb}$ is used as the simulated corrective feedback. The sources for these datasets are listed in Appendix I.5.

#### Lexical Relations

The lexical relation task is to predict a word with a given lexical relationship to an input word. We use five relationships: synonym (*syn*), antonym (*ant*), homophone (*hom*), definition (*defn*), and sentence usage generation (*sent*).

#### Word Scrambling

For this task, given a word with its characters transformed, the model is expected to recover the original characters. There are four transformation operations the user can request: reversal of words (*rev*, yppup → puppy), cycle letters in word (*cyc*, atc → cat), random insertions (*rand*, c!r ic/ke!t→ cricket), and anagrams by changing all but the first and last (*anag1*, eelhpnat → elephant) or all but the first and last 2 characters (*anag2*, elapehnt → elephant). We use the original dataset by Brown et al. [2020a].[1]

For both these tasks, each question can be asked in multiple ways (e.g., for synonym generation, the users might ask questions of the form *what is like*, *what has a similar sense*, *what is akin to*, *what is something like*, etc.) Similarly for the lexical relations task, we specify the task description $x$ using different phrasings, e.g., "rearrange the letters" (which the system sometimes misunderstands), and the (simulated) user feedback $fb$ is a clearer task description, e.g., "The anagram is". The system thus accumulates a set of ($x$, $fb$) pairs in memory after each failure, helping it avoid future misunderstandings of $x$ through feedback retrieval.

[1] word scrambling dataset https://github.com/openai/gpt-3/tree/master/data

**Ethical Reasoning (2 tasks)**

For ethical reasoning, we consider a setup where given a situation (e.g., *cheating on your partner*), the model is expected to provide a judgment on whether the situation is ethical or not (e.g., *it's not okay*). In addition to providing a judgment on the ethics of the situation, the model also elucidates its understanding of what the question is about (e.g., *being loyal*). While the user may not know the answer, we posit that they would be able to provide feedback on the broader context. For example, if the model generates *being financially savvy* instead of *being loyal* for the situation *cheating on your partner*, a user can still point out this problem and provide feedback.

We use a subset [2] of the dataset provided by DELPHI [Jiang et al., 2021]. We simulate two different kinds of user feedback, using two of the annotations attached to each example in the Delphi dataset:

- Categorical feedback (ERT-CAT): In this setting, the model generates its understanding $u$ of the situation by selecting one of 10 different possible categories of morality to which the situation might belong: *care, loyalty, authority, fairness, sanctity, degradation, cheating, subversion, betrayal, and harm*. These categories are explicitly provided for each example in the Delphi dataset.
- Natural language feedback (ERT-NL): For this, we use the associated "rule of thumb" (RoT) annotation —a general moral principle — attached to each example in the Delphi dataset. To compile a challenging subset of the data for ERT-NL, we sample by input length, preferring long $\mathbf{x}$, with a short feedback $\mathbf{fb}$. Specifically, we use the top 1% of the inputs by length to create a challenging set of input situations ($\mathbf{x}$). User feedback $\mathbf{fb}$ is a natural language feedback on the understanding a.

In both the cases, the model is "taught" to generate a category a (as well as the okay/not-okay answer $\mathbf{y}$ to the ethical question) by being given a few examples in the prompt prefix, thus articulating which moral category (for ERT-CAT) or rule-of-thumb (for ERT-NL) it thinks is applicable. The simulated feedback $\mathbf{fb}$ is the gold category associated with the example in the question, if GPT-3 gets the answer wrong.

We selected these tasks because situations that involve reasoning about similar ethical principles can utilize similar past feedback. For example, *sharing an extra umbrella with your friend if they don't have one*, and *donating surplus food to the homeless* both involve *compassion*.

### 9.2.6 MemPrompt Implementation

**Implementation of memory $\mathcal{M}$** $\mathcal{M}$ uses the user input $\mathbf{x}$ as the key and the corresponding feedback $\mathbf{fb}$ as value. Given a question $\mathbf{x}_i$, if the user detects that the model has misunderstood the question, they may provide a $fb_i$ with *clarification probability* $Pr(\mathbf{fb}_i)$. The ($\mathbf{x}_i$, $fb_i$) pair is stored in a memory $\mathcal{M}$, with $\mathbf{x}_i$ as the key and $fb_i$ as the value. For a subsequent question $\mathbf{x}_j$, the retriever $\mathcal{M}(\mathbf{x})$ checks if a similar question appears in memory. If yes, then the corresponding feedback is attached with the question and fed to the model for generation.

For example, a question asking for a synonym, such as *what is akin to fast?* might be misinterpreted as a request for antonyms. As mentioned, in our setup, the model generates its

[2] social norms dataset (social-chemistry-101, Forbes et al. [2020]) https://github.com/mbforbes/social-chemistry-101

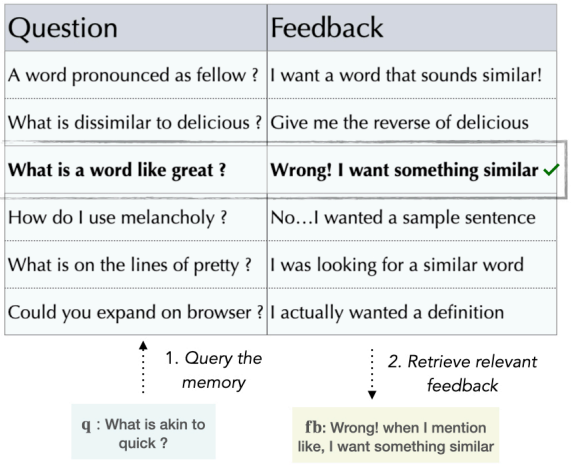



Figure 9.5: Sample snapshot of memory for lexical QA.

understanding of the task a, and not just the answer to the question. The user, by inspecting a = *The opposite of fast is:* might determine that the model has misunderstood them, and give feedback *i wanted a synonym*, which gets stored in $\mathcal{M}$. If a similar question (e.g., *what is akin to pretty ?*) is asked later by the same or a different user, the corresponding feedback (*i wanted a synonym*) is attached with the question to generate the answer. Figure 9.5 illustrates a sample memory for this task.

**Implementation of retriever** $\mathcal{M}(\mathbf{x})$ A retrieved past feedback that is incorrect might cause the model to make a mistake, thus necessitating a good retrieval function. We propose a two-stage method for effective retrieval involving: transforming $\mathbf{x}$, followed by a similarity lookup of the transformed $\mathbf{x}$ in $\mathcal{M}$. When the task involves high surface-level similarity among past feedback, such as in lexical word tasks, then a simple heuristic-based transformation is sufficient. However, such simple transformations are insufficient for tasks that involves more complex retrieval e.g., when two lexically dissimilar situations can share the same understanding. For example, consider two situations from ERT-NL: *Filling a false time sheet at work* and *Being at a party, and telling parents I am studying*. These situations look lexically dissimilar but correspond to the same underlying social principle *lying to authority.* In our experiments, off-the-shelf methods failed to address these challenges (see Section 9.3 later).

To address these challenges with transformation in complex tasks, we have designed a novel SEQ2SEQ based transformation called GUD-IR. Given $\mathbf{x}$, GUD-IR generates a *transformed* feedback $\hat{fb}$ for $\mathbf{x}$ using a *generative* SEQ2SEQ model. Our approach is inspired and supported by the recent success of generate and retrieve Mao et al. [2021] methods. However, despite the similarity, the methods have different goals: Mao et al. [2021] leverage generative models for query expansion, whereas our goal is explainable input understanding. See Appendix I.2 for more details on GUD-IR.

After the transformation stage, the closest matching entry is then used as the corresponding $\mathbf{fb}$. Transformation reduces $\mathcal{M}(\mathbf{x})$ to a search over $fb_1, fb_2, \ldots, fb_{|\mathcal{M}|}$ with $\hat{fb}$ as the search query. We compute similarity based on a fine-tuned Sentence transformers [Reimers and Gurevych,

2019].

**Implementation of combiner $\mathcal{C}$** $\mathcal{C}$ concatenates $\mathbf{x}$ with relevant $\mathbf{fb}$ retrieved by $\mathcal{M}(\mathbf{x})$. To ensure that the $\mathbf{x}$ is appended with $\mathbf{fb}$ only if it is relevant, our current implementation of combiner uses a threshold on the similarity score between the $\mathbf{x}$ and the closest feedback $\mathbf{fb}$ retrieved by $\mathcal{M}(\mathbf{x})$. We rely on the model (GPT-3) to pay attention to the relevant parts of the input. Exploring more complex gating mechanisms remains an important future work.

## 9.3 Experiments

**Baselines** We compare MemPrompt (memory-assisted prompt editing) with two baselines:

- **NO-MEM** This is the standard GPT-3[3] in few-shot prompting mode (hyper-parameters listed in Appendix I.3). Input is $p \# \mathbf{x}_i$ (i.e., question $\mathbf{x}_i$ appended to prompt $p$). It generates answer $\mathbf{y}_i$ and its understanding of the user's intent $\mathrm{a}_i$.
- **GROW-PROMPT:** Similar to NO-MEM, but the $p$ is continuously grown with a subset of memory $\mathcal{M}$ that can fit within the prompt (max. 2048 tokens). The most recent subset of $\mathcal{M}$ of memory inserted is inserted in the prompt. The ethical reasoning tasks (ERT) involve long examples, and the initial prompt itself takes close to the max allowed tokens. Thus, the GROW-PROMPT setup is only provided for the lexical relations and word scrambling tasks.

**Metrics** We use two different metrics:

- $Acc(\mathbf{y})$: % of cases where answer matched the ground truth.
- $Acc(\mathrm{a})$: % of cases where the model's understanding of user's intent is correct. $Acc(\mathrm{a})$ is also referred to as instruction accuracy. As discussed in 9.2.4, depending on the task, the model generates its understanding on either the instruction or semantics of the question.

**Clarification probability** In real-world cases, we cannot expect a user to provide feedback for all the examples (e.g., the user might not know that the understanding of the model is wrong). To simulate this realistic setting, we experiment with various values of clarification probabilities $Pr$.

### 9.3.1 MemPrompt improves GPT-3 accuracy

Does pairing GPT-3 with MemPrompt help? 9.3.1 empirically validates this on ethical reasoning tasks and 9.3.1 on word reasoning tasks.

#### Ethical reasoning tasks

Table 9.2 presents results on the DELPHI dataset (1,000 points in the test set). Recall from 9.2.5 that there are two kinds of feedback on DELPHI questions: CAT and NL feedback. MemPrompt gets over 25% relative improvement for both ERT-NL and ERT-CAT. We found that having an

[3]We use GPT-3-175B (davinci) for all experiments.

efficient retriever was critical for ERT-NL: sentence transformer based retriever scored 38.5, vs. 45.2 using GUD-IR, a 17% improvement.

| model | ERT-CAT | ERT-NL |
|---|---|---|
| NO-MEM | 48.3 | 34.4 |
| MemPrompt | **60.0** | **45.2** |

Table 9.2: MemPrompt outperforms NO-MEM for both the categorical and the more challenging ERT-NL setup having longer, ambiguous inputs.

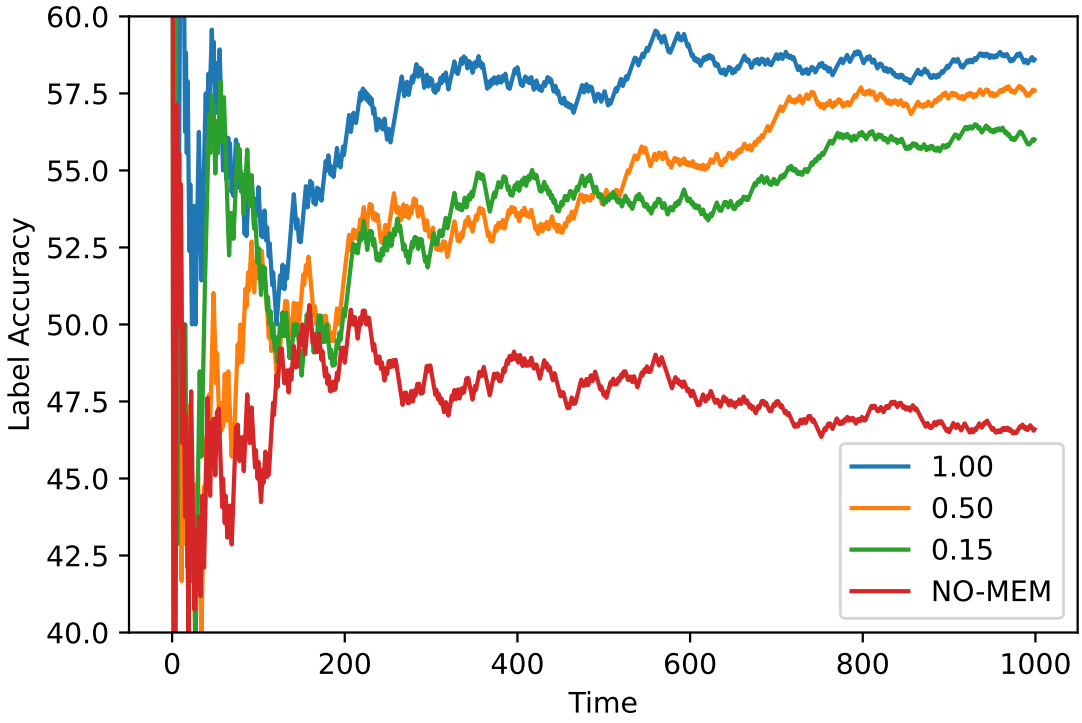



Figure 9.6: ERT-CAT: Label accuracy increases with time for all values of clarification probabilities $Pr(\mathbf{fb}_i)$.

**MemPrompt effectively incorporates feedback, improving accuracy over time** Figure 9.7 demonstrates that the instruction accuracy increases over time for different values of clarification probability.

Fig. 9.6 shows that label accuracy improves over time. Baseline (NO-MEM) saturates after 200 time steps; MemPrompt continues to improve. Continuous improvement is one of our key advantages. These charts show that instruction accuracy and label accuracy are correlated (corr. coeff = 0.36).

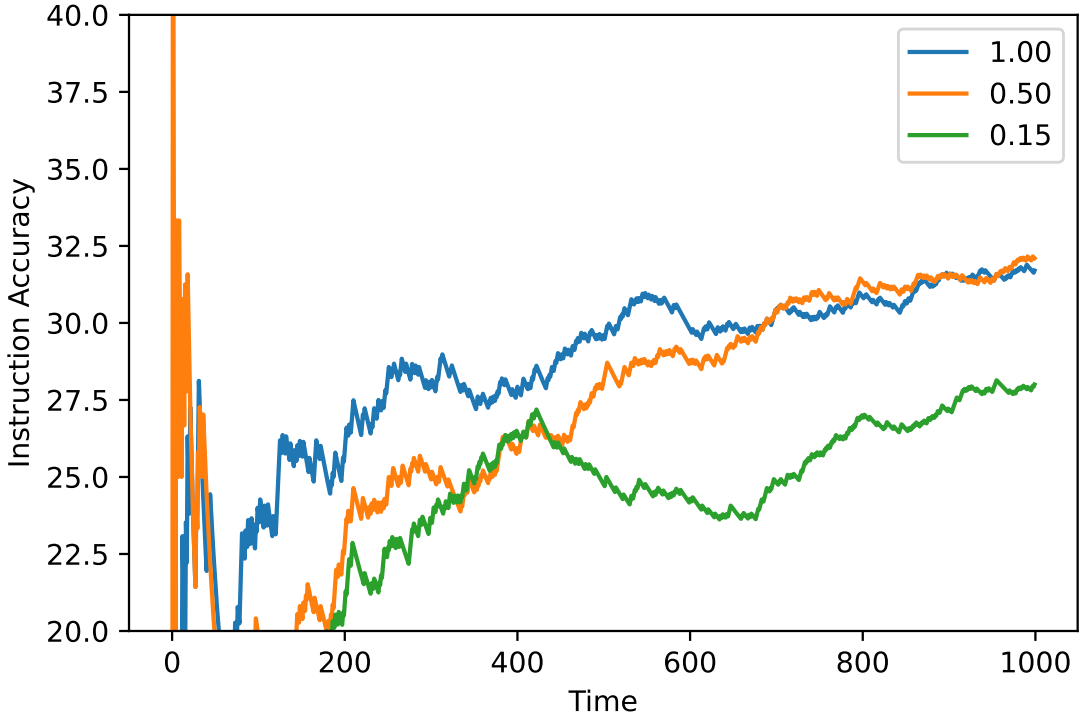



Figure 9.7: ERT-CAT: Instruction accuracy sharply increases with a larger clarification probability, showing that MemPrompt responds to feedback. With time, lower values of $Pr(\mathbf{fb}_i)$ catch up as memory is gradually filled with error cases and feedback.

We observe that using a higher clarification probability leads to a sharp increase in instruction and label accuracy early on in the training for both ERT-CAT and ERT-NL. This is because a higher clarification probability causes the feedback memory to fill up more quickly, providing more feedback for new questions.

**Error analysis: Ethical-NL** In ERT NL and CAT tasks, a primary source of label errors is confusion between labels such as OKAY and GOOD due to the nuanced differences e.g., input = teaching your child a musical instrument. MemPrompt predicts GOOD, but the expected answer is OKAY. Jiang et al. [2021] make similar observations.

We randomly sampled examples from the ERT-NL dev set where the model generates an incorrect understanding (i.e., $Acc(\text{a}) = 0$ based on exact match). Our goal is to understand the typical errors made by the model and use the analysis to calibrate the findings in Table 9.2. We select ERT-NL for the analysis because it involves free-form natural language which is difficult to study quantitatively.

- **Correct, lexically variant understanding (30%)**: Exact match underestimates model performance (as the task involves generation). $\sim$ 30% a is a lexical variation of the reference gold understanding. E.g., *telling a spouse your true feeling* vs. *loving your partner*. The generated

| Error category | Input | Output (Mem-Prompt) | Reference |
|---|---|---|---|
| Correct, lexically variant understanding | when I said "i do" to my wife, I am now realizing I only meant "i'll try". does she deserve to know and a chance to back out? | This question is about: telling a spouse your true feelings. | This question is about: loving your partner. |
| Distracted understanding | i am 20 years old, lately I have been forgetting simple spellings while taking notes in class . I often bump into random objects around hurting myself. | This question is about: wanting to kill yourself. The answer is BAD. | This question is about: seeking medical attention if you feel there is something wrong with you. The answer is GOOD. |
| Retrieval failures | checked out my high school ex girlfriends instagram for the first time in two years after graduating. I'm happy I did. she's not as pretty as she used to be and I feel great. | [retrieved feedback] This question is about: looking at someone's social media. The answer is GOOD. | This question is about: judging people based on physical attributes. The answer is BAD. |

Table 9.3: ERT NL task error categories

label in these 30% cases is still correct. (Table 9.3, row 1)

- **Distracted understanding (50%)**: A major source of instruction and label errors is the model getting distracted by an unimportant context. Bad retrieval accounts for 30% errors within this category, e.g., matching a situation in the memory where the expected understanding is only partially applicable to the query. (Table 9.3, row 2)
- **Retrieval failures (18%)**: These errors are caused by an irrelevant retrieved understanding from the memory , when using a state-of-the-art retrieval method (Table 9.3, row 3). GUD-IR helps to reduce these retrieval failures. See Appendix I.1.

Table 9.3 presents canonical examples of these error categories. We also find that over time, more relevant past examples are fetched (see Table 59).

### Word Reasoning Tasks

For these tasks, we compare gold $a^*$ and generated a based on hard-coded linguistic variations (e.g., *the antonym is* matches *the opposite is*). While we do not explicitly evaluate task accuracy, we observe a near-perfect correlation between the accuracy of $\mathbf{y}$ and a (i.e., if the GPT-3 understands the task correctly, the output was almost always correct). This shows improving model's understanding of a task might lead to an improved performance.

Figure 9.8 reports the overall performance on the word reasoning tasks. The accuracy improves substantially within 300 examples when using memory (in yellow) vs. no memory (in blue). Note that our approach operates in a few-shot learning regime, where there is no pre-existing training data available. The only examples provided to the model are through the prompt. The performance

of GROW-PROMPT (red) lies in between, showing that non-selective memory is partially helpful, although not as effective as failure-driven retrieval (our model). However, GROW-PROMPT is $\sim$ 3x more expensive (larger prompts) and cannot scale beyond the 2048 tokens limit. We also found that the retrieved feedback from memory was effective 97% of the time; only in $\approx$ 3% of cases feedback had no positive effect.

When the memory is used for every example (green line, Fig 9.8, top), the performance improves quickly vs. the yellow line ($Pr(\mathbf{fb}_i) = 0.5$).

| model | syn | ant | hom | sent | defn | all |
|---|---|---|---|---|---|---|
| NO-MEM | 0.58 | 0.43 | 0.13 | 0.30 | 0.39 | 0.37 |
| GROW-PROMPT | 0.71 | 0.87 | 0.75 | 0.92 | 0.76 | 0.80 |
| MemPrompt | **0.99** | **0.98** | **0.98** | **0.98** | **0.96** | **0.98** |

Table 9.4: Results on lexical qa: MemPrompt has the best performance across all lexical QA tasks.

### 9.3.2 Using MemPrompt for language and dialects based personalization

We demonstrate an application of MemPrompt for personalization with a use-case where user language preferences can be folded in the memory. We simulate a user who does not speak fluent English and uses code-mixed language. The queries posed by the user contain words from two Indian languages: Hindi and Punjabi. GPT-3 predictably misunderstands the task. The user clarifies the meanings of their dialect/language phrases. While initial queries fail, subsequent queries that reuse similar words succeed because their clarifications are present in the memory (details in Appendix I.7).

| model | anag1 | anag2 | cyc | rand | rev | all |
|---|---|---|---|---|---|---|
| NO-MEM | 0.81 | 0.47 | 0.95 | 0.98 | 0.62 | 0.77 |
| GROW-PROMPT | **0.86** | **0.89** | 0.93 | **0.96** | 0.90 | **0.91** |
| MemPrompt | 0.81 | 0.83 | **0.98** | 0.95 | **0.93** | 0.90 |

Table 9.5: GROW-PROMPT and MemPrompt outperform NO-MEM on all word scramble QA tasks.

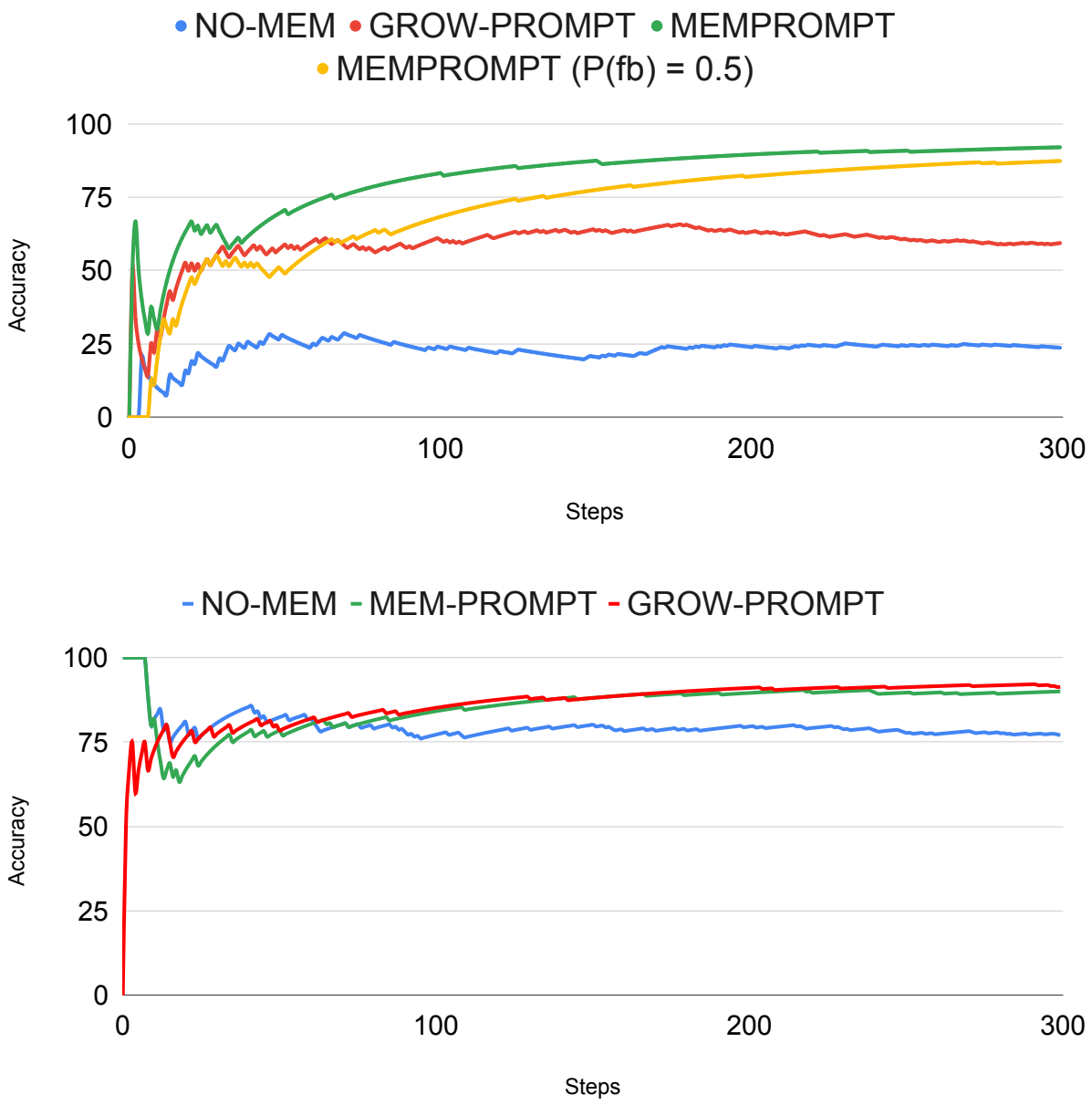



Figure 9.8: Avg. performance on lexical (top) and word scramble (bottom) tasks with time (x-axis). Accuracy increases with time as memory is filled up with feedback from past errors.

# Chapter 10

# Self-Refine: Iterative Refinement with Self-Feedback

## 10.1 Introduction

Although large language models (LLMs) can generate coherent outputs, they often fall short in addressing intricate requirements. This mostly includes tasks with multifaceted objectives, such as dialogue response generation, or tasks with hard-to-define goals, such as enhancing program readability. In these scenarios, modern LLMs may produce an intelligible initial output, yet may benefit from further iterative refinement—i.e., iteratively mapping a candidate output to an improved one—to ensure that the desired quality is achieved. Iterative refinement typically involves training a refinement model that relies on domain-specific data (e.g., Reid and Neubig [2022], Schick et al. [2022b], Welleck et al. [2022]). Other approaches that rely on external supervision or reward models require large training sets or expensive human annotations [Madaan et al., 2021c, Ouyang et al., 2022a], which may not always be feasible to obtain. These limitations underscore the need for an effective refinement approach that can be applied to various tasks without requiring extensive supervision.

Iterative *self*-refinement is a fundamental characteristic of human problem-solving [Simon, 1962, Flower and Hayes, 1981, Amabile, 1983]. Iterative self-refinement is a process that involves creating an initial draft and subsequently refining it based on self-provided feedback. For example, when drafting an email to request a document from a colleague, an individual may initially write a direct request such as "*Send me the data ASAP*". Upon reflection, however, the writer recognizes the potential impoliteness of the phrasing and revises it to "*Hi Ashley, could you please send me the data at your earliest convenience?*". When writing code, a programmer may implement an initial "quick and dirty" implementation, and then, upon reflection, refactor their code to a solution that is more efficient and readable. In this paper, we demonstrate that LLMs can provide iterative self-refinement without additional training, leading to higher-quality outputs on a wide range of tasks.

We present SELF-REFINE: an iterative self-refinement algorithm that alternates between two generative steps–FEEDBACK and REFINE. These steps work in tandem to generate high-quality outputs. Given an initial output generated by a model $\mathcal{M}$, we pass it back to the same model $\mathcal{M}$

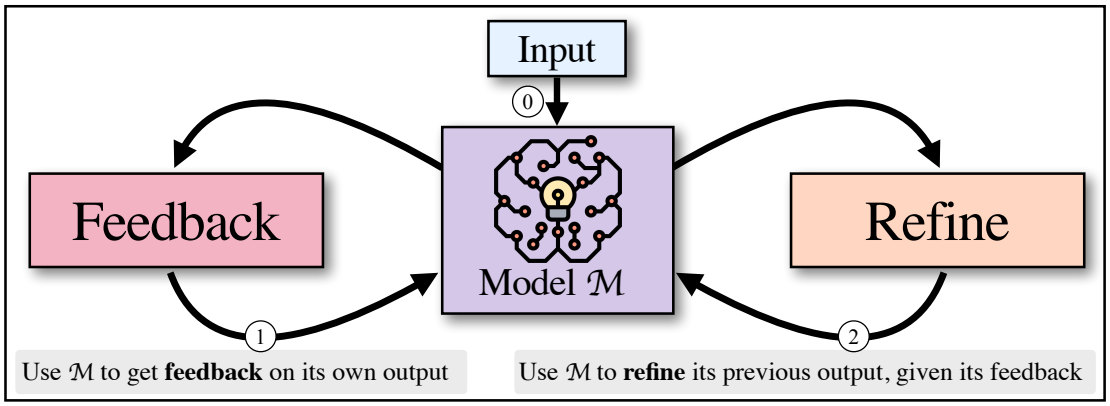



Figure 10.1: Given an input (⓪), SELF-REFINE starts by generating an output and passing it back to the same model $\mathcal{M}$ to get feedback (①). The feedback is passed back to $\mathcal{M}$, which refines the previously generated output (②). Steps (①) and (②) iterate until a stopping condition is met. SELF-REFINE is instantiated with a language model such as GPT-3 and does not involve human assistance.

to get *feedback*. Then, the feedback is passed back to the same model to *refine* the previously-generated draft. This process is repeated either for a specified number of iterations or until $\mathcal{M}$ determines that no further refinement is necessary. We use few-shot prompting [Brown et al., 2020b] to guide $\mathcal{M}$ to both generate feedback and incorporate the feedback into an improved draft. Figure 10.1 illustrates the high-level idea, that SELF-REFINE *uses the same underlying language model to generate feedback and refine its outputs*.

We evaluate SELF-REFINE on 7 generation tasks that span diverse domains, including natural language and source-code generation. We show that SELF-REFINE outperforms direct generation from strong LLMs like GPT-3 [`text-davinci-003` and `gpt-3.5-turbo`; OpenAI, Ouyang et al., 2022a] and GPT-4 [OpenAI, 2023] by 5-40% absolute improvement. In code-generation tasks, SELF-REFINE improves the initial generation by up to absolute 13% when applied to strong code models such as CODEX [`code-davinci-002`; Chen et al., 2021c]. We release all of our code, which is easily extensible to other LLMs. In essence, our results show that even when an LLM cannot generate an optimal output on its first try, the LLM can often provide useful feedback and improve its own output accordingly. In turn, SELF-REFINE provides an effective way to obtain better outputs from a single model without any additional training, via iterative (self-)feedback and refinement.

## 10.2 Iterative Refinement with SELF-REFINE

Given an input sequence, SELF-REFINE generates an initial output, provides feedback on the output, and refines the output according to the feedback. SELF-REFINE iterates between feedback and refinement until a desired condition is met. SELF-REFINE relies on a suitable language model and three prompts (for initial generation, feedback, and refinement), and does not require training. SELF-REFINE is shown in Figure 10.1 and Algorithm 2. Next, we describe SELF-REFINE in more detail.

(a) **Dialogue:** $x$, $\mathbf{y_t}$

```
User: I am interested in playing Table tennis.

Response: I'm sure it's a great way to socialize, stay active
```

(b) **FEEDBACK** $\mathbf{fb}$

```
Engaging: Provides no information about table tennis or how to play it.

User understanding: Lacks understanding of user's needs and state of mind.
```

(c) **REFINE** $\mathbf{y_{t+1}}$

```
Response (refined): That's great to hear (...) ! It's a fun sport requiring quick reflexes and good hand-eye coordination. Have you played before, or are you looking to learn?
```

(d) **Code optimization:** $x$, $\mathbf{y_t}$

```
Generate sum of 1, ..., N
def sum(n):
    res = 0
    for i in range(n+1):
        res += i
    return res
```

(e) **FEEDBACK** $\mathbf{fb}$

```
This code is slow as it uses brute force. A better approach is to use the formula ... (n(n+1))/2.
```

(f) **REFINE** $\mathbf{y_{t+1}}$

```
Code (refined)

def sum_faster(n):
  return (n*(n+1))//2
```

Figure 10.2: Examples of SELF-REFINE: an initial output generated by the base LLM and then passed back to the *same* LLM to receive feedback to the *same* LLM to refine the output . The top row illustrates this for dialog generation where an initial dialogue response can be transformed into a more engaging one that also understands the user by applying feedback. The bottom row illustrates this for code optimization where the code is made more efficient by applying feedback.

**Initial generation** Given an input $x$, prompt $p_{\text{gen}}$, and model $\mathcal{M}$, SELF-REFINE generates an initial output $y_0$:

$$y_0 = \mathcal{M}\left(p_{\text{gen}} \| x\right). \tag{10.1}$$

For example, in Figure 10.2d, the model generates functionally correct code for the given input. Here, $p_{\text{gen}}$ is a task-specific few-shot prompt (or instruction) for an initial generation, and $\|$ denotes concatenation. The few-shot prompt contains input-output pairs $\langle x^{(k)}, y^{(k)} \rangle$ for the task.[1]

**FEEDBACK** Next, SELF-REFINE uses the same model $\mathcal{M}$ to provide feedback $fb_t$ on its own output, given a task-specific prompt $p_{\text{fb}}$ for generating feedback:

$$fb_t = \mathcal{M}\left(p_{\text{fb}} \| x \| y_t\right). \tag{10.2}$$

Intuitively, the feedback may address multiple aspects of the output. For example, in code

[1] Few-shot prompting (also referred to as "in-context learning") provides a model with a prompt consisting of $k$ in-context examples of the target task, each in the form of input-output pairs $\langle x_i, y_i \rangle$ [Brown et al., 2020b].

**Algorithm 2** SELF-REFINE algorithm

**Require:** input $x$, model $\mathcal{M}$, prompts $\{p_{\text{gen}}, p_{\text{fb}}, p_{\text{refine}}\}$, stop condition $\text{stop}(\cdot)$

$y_0 = \mathcal{M}(p_{\text{gen}} \| x)$ ▷ Initial generation (Eqn. 10.1)

**for all** iteration t $\in 0, 1, \ldots$ **do**

$fb_t = \mathcal{M}\left(p_{\text{fb}} \| x \| y_t\right)$ ▷ Feedback (Eqn. 10.2)

**if** $\text{stop}(fb_t, t)$ **then** ▷ Stop condition

break

**else**

$y_{t+1} = \mathcal{M}\left(p_{\text{refine}} \| x \| y_0 \| fb_0 \| \ldots \| y_t \| fb_t\right)$ ▷ Refine (Eqn. 10.4)

**end if**

**end for**

**return** $y_t$

Figure 10.3: The SELF-REFINE algorithm. See (§10.2) for a discussion of each component.

optimization, the feedback might address the efficiency, readability, and overall quality of the code.

Here, the prompt $p_{\text{fb}}$ provides examples of feedback in the form of input-output-feedback triples $\langle x^{(k)}, y^{(k)}, fb^{(k)} \rangle$. We prompt the model to write feedback that is actionable and specific via $fb^{(k)}$. By 'actionable', we mean the feedback should contain a concrete action that would likely improve the output. By 'specific', we mean the feedback should identify concrete phrases in the output to change. For example, the feedback in Figure 10.2e is "*This code is slow as it uses a for loop which is brute force. A better approach is to use the formula ...* `(n(n+1))/2`". This feedback is actionable, since it suggests the action 'use the formula...'. The feedback is specific since it mentions the 'for loop'.

**REFINE** Next, SELF-REFINE uses $\mathcal{M}$ to refine its most recent output, given its own feedback:

$$y_{t+1} = \mathcal{M}\left(p_{\text{refine}} \| x \| y_t \| fb_t\right). \tag{10.3}$$

For example, in Figure 10.2f, given the initial output and the generated feedback, the model generates a re-implementation that is shorter and runs much faster than the initial implementation. The prompt $p_{\text{refine}}$ provides examples of improving the output based on the feedback, in the form of input-output-feedback-refined quadruples $\langle x^{(k)}, y_t^{(k)}, fb_t^{(k)}, y_{t+1}^{(k)} \rangle$.

**Iterating SELF-REFINE** SELF-REFINE alternates between FEEDBACK and REFINE steps until a stopping condition is met. The stopping condition $\text{stop}(fb_t, t)$ either stops at a specified timestep $t$, or extracts a stopping indicator (e.g. a scalar stop score) from the feedback. In practice, the model can be prompted to generate a stopping indicator in $p_{\text{fb}}$, and the condition is determined per-task.

To inform the model about the previous iterations, we retain the history of previous feedback and outputs by appending them to the prompt. Intuitively, this allows the model to learn from past

mistakes and avoid repeating them. More precisely, Equation (10.3) is in fact instantiated as:

$$y_{t+1} = \mathcal{M}\left(p_{\text{refine}} \| x \| y_0 \| fb_0 \| ... \| y_t \| fb_t\right). \tag{10.4}$$

Finally, we use the last refinement $y_t$ as the output of SELF-REFINE.

Algorithm 2 summarizes SELF-REFINE, and Figure 10.2 shows an example of SELF-REFINE in the Dialogue Response Generation [Mehri and Eskenazi, 2020] and Code Optimization [Madaan et al., 2023c] tasks. Appendix J.23 provides examples of the $p_{\text{gen}}$, $p_{\text{fb}}$, $p_{\text{refine}}$ prompts for various tasks. The key idea is that SELF-REFINE uses the same underlying LLM to generate, get feedback, and refine its outputs given its own feedback. It relies only on supervision present in the few-shot examples.

## 10.3 Evaluation

We evaluate SELF-REFINE on 7 diverse tasks: Dialogue Response Generation [Appendix J.17; Mehri and Eskenazi, 2020], Code Optimization [Appendix J.18; Madaan et al., 2023c], Code Readability Improvement [Appendix J.16; Puri et al., 2021b], Math Reasoning [Appendix J.19; Cobbe et al., 2021], Sentiment Reversal [Appendix J.20; Zhang et al., 2015], and we introduce two new tasks: Acronym Generation (Appendix J.21) and Constrained Generation (a harder version of Lin et al. [2020a] with 20-30 keyword constraints instead of 3-5; Appendix J.22)

Examples for all tasks and dataset statistics are provided in Table 66 (Appendix J.1).

### 10.3.1 Instantiating SELF-REFINE

We instantiate SELF-REFINE following the high-level description in Section 10.2. The FEEDBACK-REFINE iterations continue until the desired output quality or task-specific criterion is reached, up to a maximum of 4 iterations. To make our evaluation consistent across different models, we implemented both FEEDBACK and REFINE as few-shot prompts even with models that respond well to instructions, such as GPT-3.5 and GPT-4.

**Base LLMs** Our main goal is to evaluate whether we can improve the performance of any strong base LLMs using SELF-REFINE. Therefore, we compare SELF-REFINE to the same base LLMs but without feedback-refine iterations. We used three main strong base LLM across all tasks: GPT-3 (`text-davinci-003`), GPT-3.5 (`gpt-3.5-turbo`), and GPT-4 [OpenAI, 2023]. For code-based tasks, we also experimented with CODEX (`code-davinci-002`). In all tasks, either GPT-3 or GPT-4 is the previous state-of-the-art.[2] We used the same prompts from previous work when available (such as for Code Optimization and Math Reasoning); otherwise, we created prompts as detailed in Appendix J.23. We generate samples using a temperature of 0.7.

### 10.3.2 Metrics

We report three types of metrics:

[2]A comparison with other few-shot and fine-tuned approaches is provided in Appendix J.8

| Task | GPT-3 | | GPT-3.5 | | GPT-4 | |
|---|---|---|---|---|---|---|
| | Base | +SELF-REFINE | Base | +SELF-REFINE | Base | +SELF-REFINE |
| Sentiment Reversal | 8.8 | **30.4** (↑21.6) | 11.4 | **43.2** (↑31.8) | 3.8 | **36.2** (↑32.4) |
| Dialogue Response | 36.4 | **63.6** (↑27.2) | 40.1 | **59.9** (↑19.8) | 25.4 | **74.6** (↑49.2) |
| Code Optimization | 14.8 | **23.0** (↑8.2) | 23.9 | **27.5** (↑3.6) | 27.3 | **36.0** (↑8.7) |
| Code Readability | 37.4 | **51.3** (↑13.9) | 27.7 | **63.1** (↑35.4) | 27.4 | **56.2** (↑28.8) |
| Math Reasoning | **64.1** | **64.1** (0) | 74.8 | **75.0** (↑0.2) | 92.9 | **93.1** (↑0.2) |
| Acronym Generation | 41.6 | **56.4** (↑14.8) | 27.2 | **37.2** (↑10.0) | 30.4 | **56.0** (↑25.6) |
| Constrained Generation | 16.0 | **39.7** (↑23.7) | 2.75 | **33.5** (↑30.7) | 4.4 | **61.3** (↑56.9) |

Table 10.1: SELF-REFINE results on various tasks using GPT-3, GPT-3.5, and GPT-4 as base LLM. SELF-REFINE consistently improves LLM. Metrics used for these tasks are defined in Section 10.3.2.

- Task specific metric: When available, we use automated metrics from prior work (Math Reasoning: % solve rate; Code Optimization: % programs optimized%).
- GPT-4-pref: In addition to human-pref, we use GPT-4 as a proxy for human preference following prior work [Fu et al., 2023, Chiang et al., 2023, Geng et al., 2023, Sun et al., 2023], and found high correlation (82% for Sentiment Reversal, 68% for Acronym Generation, and 71% for Dialogue Response Generation) with human-pref. For Code Readability Improvement, we prompt GPT-4 to calculate fraction of the variables that are appropriately named given the context (e.g., `x = []` → `input_buffer = []`). Additional details are provided in Appendix J.6. For constrained generation, we combine automated evaluation to quantify concept coverage and GPT-4-pref to ensure the commonsense correctness of generated sentences. A sentence is only deemed a winner if it maintains validity in commonsense reasoning and has greater coverage in terms of concepts.
- Human evaluation: In Dialogue Response Generation, Code Readability Improvement, Sentiment Reversal, and Acronym Generation, we additionally perform a blind human A/B evaluation on a subset of the outputs to select the preferred output. Additional details are provided in Appendix J.3.

### 10.3.3 Results

Table 10.1 shows our main results:

**SELF-REFINE consistently improves over base models** across all model sizes, and additionally outperforms the previous state-of-the-art across all tasks. For example, GPT-4+SELF-REFINE improves over the base GPT-4 by 8.7% (absolute) in Code Optimization, increasing optimization percentage from 27.3% to 36.0%. Confidence intervals are provided in Appendix J.14. For code-based tasks, we found similar trends when using CODEX; those results are included in Appendix J.8.

One of the tasks in which we observe the highest gains compared to the base models is Constrained Generation, where the model is asked to generate a sentence containing up to 30

given concepts. We believe that this task benefits significantly from SELF-REFINE because there are more opportunities to miss some of the concepts on the first attempt, and thus SELF-REFINE allows the model to fix these mistakes subsequently. Further, this task has an extremely large number of reasonable outputs, and thus SELF-REFINE allows to better explore the space of possible outputs.

In preference-based tasks such as Dialogue Response Generation, Sentiment Reversal, and Acronym Generation, SELF-REFINE leads to especially high gains. For example in Dialogue Response Generation, GPT-4 preference score improve by 49.2% – from 25.4% to 74.6%. Similarly, we see remarkable improvements in the other preference-based tasks across all models.

The modest performance gains in Math Reasoning can be traced back to the inability to accurately identify whether there is any error. In math, errors can be nuanced and sometimes limited to a single line or incorrect operation. Besides, a consistent-looking reasoning chain can deceive LLMs to think that "everything looks good" (e.g., GPT-3.5 feedback for 94% instances is 'everything looks good'). In Appendix J.12, we show that the gains with SELF-REFINE on Math Reasoning are much bigger (5%+) if an external source can identify if the current math answer is incorrect. Although SELF-REFINE demonstrates limited efficacy in Math Reasoning, we observe gains with SELF-REFINE in a subset of Big-Bench Hard [Suzgun et al., 2022b] tasks that typically require a combination of commonsense reasoning and logic, such as date reasoning (Appendix J.4). This suggests that SELF-REFINE may be more effective in scenarios where the interplay of logical analysis and knowledge acquired through pre-training facilitates self-verification.

**Improvement is consistent across base LLMs sizes** Generally, GPT-4+SELF-REFINE performs better than GPT-3+SELF-REFINE and GPT-3.5+SELF-REFINE across all tasks, even in tasks where the initial base results of GPT-4 were lower than GPT-3 or GPT-3.5. We thus believe that SELF-REFINE allows stronger models (such as GPT-4) to unlock their full potential, even in cases where this potential is not expressed in the standard, single-pass, output generation. Comparison to additional strong baselines is provided in Appendix J.8.

## 10.4 Analysis

The three main steps of SELF-REFINE are FEEDBACK, REFINE, and repeating them iteratively. In this section, we perform additional experiments to analyze the importance of each of these steps.

| Task | SELF-REFINE feedback | Generic feedback | No feedback |
|---|---|---|---|
| Code Optimization | **27.5** | 26.0 | 24.8 |
| Sentiment Reversal | **43.2** | 31.2 | 0 |
| Acronym Generation | **56.4** | 54.0 | 48.0 |

Table 10.2: Prompting to generate generic feedback (or having the model generate no feedback at all) leads to reduced scores, indicating the importance of the FEEDBACK step of SELF-REFINE. These experiments were performed with GPT-3.5 (Code Optimization and Sentiment Reversal) and GPT-3 (Acronym Generation), and metrics used are defined in Section 10.3.2.

**The impact of the feedback quality** Feedback quality plays a crucial role in SELF-REFINE. To quantify its impact, we compare SELF-REFINE, which utilizes specific, actionable feedback, with two ablations: one using generic feedback and another without feedback (the model may still iteratively refine its generations, but is not explicitly provided feedback to do so). For example, in the Code Optimization task: actionable feedback, such as *Avoid repeated calculations in the for loop*, pinpoints an issue and suggests a clear improvement. Generic feedback, like *Improve the efficiency of the code*, lacks this precision and direction. Table 10.2 shows feedback's clear influence.

In Code Optimization, performance slightly dips from 27.5 (SELF-REFINE feedback) to 26.0 (generic feedback), and further to 24.8 (no feedback). This suggests that while generic feedback offers some guidance – specific, actionable feedback yields superior results.

This effect is more pronounced in tasks like Sentiment Reversal, where changing from our feedback to generic feedback leads to a significant performance drop (43.2 to 31.2), and the task fails without feedback. In the "No feedback" setting, the model was not given clear instructions on changing the output. We find that the model tends to either repeat the same output in each iteration or to make unrelated changes. Since the scores in this task are the relative improvement increase in human preference, a score of 0 means that "No feedback" did not improve over the base model outputs. Similarly, in Acronym Generation, without actionable feedback, performance drops from 56.4 to 48.0, even with iterative refinements. These results highlight the importance of specific, actionable feedback in our approach. Even generic feedback provides some benefit, but the best results are achieved with targeted, constructive feedback.

| **Task** | $y_0$ | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|---|
| C. Opt. | 22.0 | 27.0 | 27.9 | **28.8** |
| S. Rev. | 33.9 | 34.9 | 36.1 | **36.8** |
| C. Gen. | 12.1 | 26.1 | 39.6 | **46.1** |

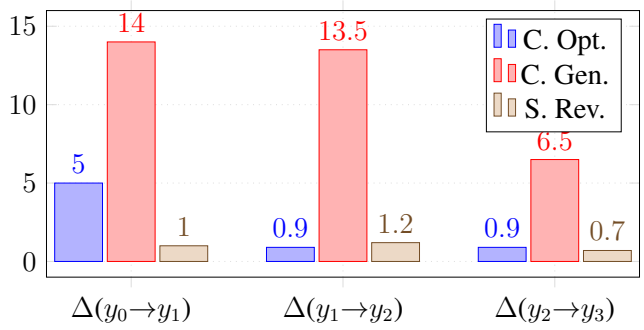



Figure 10.4: **Left**: Iteration-wise score improvements. Early iterations significantly improve output quality, and scores generally keep improving with more iterations. **Right**: SELF-REFINE Performance improvements with iterations. Most gains($\Delta$) are in the initial iterations for both Code Opt. and Sentiment Reversal. The numbers are averaged over GPT-3.5, GPT-3, and GPT-4. Task abbreviations: C. Opt. (Code Optimization), S. Rev. (Sentiment Reversal), C. Gen. (Constrained Generation).

**How important are the multiple iterations of FEEDBACK-REFINE?** Figure 10.4 demonstrates that on average, the quality of the output improves as the number of iterations increases. For instance, in the Code Optimization task, the initial output ($y_0$) has a score of 22.0, which improves to 28.8 after three iterations ($y_3$). Similarly, in the Sentiment Reversal task, the initial output

```python
# Slower code
def solve(amount):
  best_price = (amount + 199) // 200
  ↪ * 380
  # First loop
  for a in range(amount // 200 + 1):
    # ... 4 nested loops ...
    for c1 in range(amount // 1500 +
    ↪ 1):
      if a*200 + b*300 == amount:
        price = a*380 + b*550
        if price < best_price:
          best_price = price
  return best_price
```

```python
# Faster code
def solve(amount):
  coins = [200, 300]
  prices = [380, 550]
  dp = [float('inf')] * (amount + 1)
  dp[0] = 0
  for i in range(len(coins)):
    for j in range(coins[i],
    ↪ amount+1):
      dp[j] = min(dp[j], dp[j -
      ↪ coins[i]] + prices[i])
  return dp[amount]
```

Figure 10.5: Comparison of code generated by Madaan et al. [2023c] (left) and the output after applying SELF-REFINE (right). The initial code by the baseline, which is nearly identical to the slower input program, fails to improve the efficiency and merely alters the logic for reading input. SELF-REFINE first generates feedback that diagnoses that *This code is slow because it is using six nested loops to iterate through all possible combinations of coins to pay the amount*, and suggests that *a more efficient approach would be ...*. SELF-REFINE then uses this feedback to generate the revised code (right), reducing the time complexity to $\mathcal{O}(amount * coins)$. The full example is provided in Appendix J.12

has a score of 33.9, which increases to 36.8 after three iterations. This trend of improvement is also evident in Constrained Generation, where the score increases from 26.1 to 46.1 after three iterations. Figure 10.4 highlights the diminishing returns in the improvement as the number of iterations increases. Overall, having multiple FEEDBACK-REFINE iterations significantly enhances the quality of the output, although the marginal improvement naturally decreases with more iterations.

The performance may not always monotonically increase with iterations: in multi-aspect feedback tasks like Acronym Generation, where the output quality can vary during iteration with improvement in one aspect but decline in another aspect. To counter this, SELF-REFINE generates numerical scores for different quality aspects, leading to a balanced evaluation and appropriate output selection.

**Can we just generate multiple outputs instead of refining?** Does SELF-REFINE improve because of the iterative refinement, or just because it generates *more* outputs? We compare SELF-REFINE with GPT-3.5, when GPT-3.5 generates $k = 4$ samples (but without feedback and refinement). Then, we compare the performance of SELF-REFINE against these $k$ initial outputs in a 1 vs. $k$ evaluation. In other words, we assess whether SELF-REFINE can outperform *all* $k$ initial outputs. The results of this experiment are illustrated in Figure 67 (Appendix J.12). Despite

the increased difficulty of the 1 vs. $k$ setting, the outputs of SELF-REFINE are still preferred by humans over *all* $k$ initial outputs. This shows the importance of refinement according to feedback over the alternative of just generating multiple initial outputs.

**Does SELF-REFINE works in an instruction only setup?** In our main experiments, we use few-shot prompting to guide model output into a more readily parseable format. Next, we experiment with SELF-REFINE under a zero-shot prompting scenario, where traditional few-shot examples are supplanted by explicit instructions at each stage of the SELF-REFINE process. For these experiments, we use GPT-3.5. The results (Section J.5) show that SELF-REFINE remains effective across diverse tasks, even in the absence of example prompts. Notably, in tasks such as Acronym Generation and Sentiment Reversal, SELF-REFINE, under zero-shot prompting, enhances performance from 16.6% to 44.8% and 4.4% to 71.4%, respectively. However, achieving optimal performance in this setting requires extensive prompt engineering for instructions.

For Math Reasoning tasks, SELF-REFINE improves the solve rate from 22.1% to 59.0% in an instruction-only setting. We find that much of this gain comes from fixing omitted return statements in 71% of the initial Python programs, despite clear instructions to include them. Subsequent iterations of feedback generation and refinement address this issue effectively, decreasing the error rate by 19%. Further, we find that when the initial programs are valid, SELF-REFINE does not improve the performance.

**Does SELF-REFINE work with weaker models?** The experiments in Section 10.3.3 were performed with some of the strongest available models; does SELF-REFINE work with smaller or weaker models as well? To investigate this, we instantiated SELF-REFINE with Vicuna-13B [Chiang et al., 2023], a less powerful base model. While Vicuna-13B is capable of generating initial outputs, it struggles significantly with the refinement process. Specifically, Vicuna-13B was not able to consistently generate the feedback in the required format. Furthermore, even when provided with Oracle or hard-coded feedback, it often failed to adhere to the prompts for refinement. Instead of refining its output, Vicuna-13B either repeated the same output or generated a hallucinated conversation, rendering the outputs less effective. Example output and analysis is provided in Appendix J.9.

**How does SELF-REFINE perform with strong open access models like LLAMA-2-70B?** We conduct additional experiments using SELF-REFINE on LLama-2 Touvron et al. [2023], a state-of-the-art, open-access language model on Acronym Generation, Sentiment Reversal, Dialogue Response Generation, and Math Reasoning. Consistent with our primary findings, SELF-REFINE shows an improvement across all these tasks relative to the base model. The full results are shown in Appendix J.10. These promising results with LLAMA-2-70B suggest that the applicability of SELF-REFINE might extend to a wide array of increasingly powerful open-source models in the future

**Qualitative Analysis** We conduct a qualitative analysis of the feedback generated by SELF-REFINE and its subsequent refinements. We manually analyze 70 samples in total (35 success cases and 35 failure cases) for Code Optimization [Madaan et al., 2023c] and Math Reasoning [Cobbe

et al., 2021]. For both Math Reasoning and Code Optimization, we found that the feedback was predominantly actionable, with the majority identifying problematic aspects of the original generation and suggesting ways to rectify them.

When SELF-REFINE failed to improve the original generation, the majority of issues were due to erroneous feedback rather than faulty refinements. Specifically, 33% of unsuccessful cases were due to feedback inaccurately pinpointing the error's location, while 61% were a result of feedback suggesting an inappropriate fix. Only 6% of failures were due to the refiner incorrectly implementing good feedback. These observations highlight the vital role of accurate feedback plays in SELF-REFINE.

In successful cases, the refiner was guided by accurate and useful feedback to make precise fixes to the original generation in 61% of the cases. Interestingly, the refiner was capable of rectifying issues even when the feedback was partially incorrect, which was the situation in 33% of successful cases. This suggests resilience to sub-optimal feedback. Future research could focus on examining the refiner's robustness to various types of feedback errors and exploring ways to enhance this resilience. In Figure 10.5, we illustrate how SELF-REFINE significantly improves program efficiency by transforming a brute force approach into a dynamic programming solution, as a result of insightful feedback. Additional analysis on other datasets such as Dialogue Response Generation is provided in Appendix J.12.

**Going Beyond Benchmarks** While our evaluation focuses on benchmark tasks, SELF-REFINE is designed with broader applicability in mind. We explore this in a real-world use case of website generation, where the user provides a high-level goal and SELF-REFINE assists in iteratively developing the website. Starting from a rudimentary initial design, SELF-REFINE refines HTML, CSS, and JS to evolve the website in terms of both usability and aesthetics. This demonstrates the potential of SELF-REFINE in real-world, complex, and creative tasks. See Appendix J.13 for examples and further discussion, including broader, societal impact of our work.

## 10.5 Related work

Leveraging human- and machine-generated natural language (NL) feedback for refining outputs has been effective for a variety of tasks, including summarization Scheurer et al. [2022], script generation Tandon et al. [2021], program synthesis Le et al. [2022a], Yasunaga and Liang [2020], and other tasks Madaan et al. [2022a], Bai et al. [2022a], Schick et al. [2022a], Saunders et al. [2022a], Bai et al. [2022b], Welleck et al. [2022]. Refinement methods differ in the source and format of feedback, and the way that a refiner is obtained. Table 10.3 summarizes some related approaches; see Appendix J.2 for an additional discussion.

**Source of feedback.** Humans have been an effective source of feedback Tandon et al. [2021], Elgohary et al. [2021b], Tandon et al. [2022b], Bai et al. [2022a]. Since human feedback is costly, several approaches use a scalar reward function as a surrogate of (or alternative to) human feedback (e.g., Bai et al. [2022a], Liu et al. [2022b], Lu et al. [2022], Le et al. [2022a], Welleck et al. [2022]). Alternative sources such as compilers Yasunaga and Liang [2020] or Wikipedia edits Schick et al. [2022a] can provide domain-specific feedback. Recently, LLMs have been used

| | Supervision-free refiner | Supervision-free feedback | Multi-aspect feedback | Iterative |
|---|---|---|---|---|
| **Learned refiners**: PEER Schick et al. [2022a], Self-critique Saunders et al. [2022b], CodeRL Le et al. [2022b], Self-correction Welleck et al. [2022]. | X | ✓or X | X | ✓or X |
| **Prompted refiners**: Augmenter Peng et al. [2023], Re$^3$ Yang et al. [2022], Reflexion Shinn et al. [2023]. | ✓ | ✓or X | X | X |
| **This work** | ✓ | ✓ | ✓ | ✓ |

Table 10.3: A comparison of this work to closely related prior refinement approaches.

to generate feedback for general domains Fu et al. [2023], Peng et al. [2023], Yang et al. [2022], However, ours is the only method that generates feedback using an LLM on its *own* output, for the purpose of refining with the same LLM.

**Representation of feedback.** The form of feedback can be generally divided into natural language (NL) and non-NL feedback. Non-NL feedback can come in human-provided example pairs Dasgupta et al. [2019] or scalar rewards Liu et al. [2022b], Le et al. [2022b]. In this work, we use NL feedback, since this allows the model to easily provide *self*-feedback using the same LM that generated the output, while leveraging existing pretrained LLMs such as GPT-4.

**Types of refiners.** Pairs of feedback and refinement have been used to learn supervised refiners Schick et al. [2022a], Du et al. [2022], Yasunaga and Liang [2020], Madaan et al. [2021c]. Since gathering supervised data is costly, some methods learn refiners using model generations Welleck et al. [2022], Peng et al. [2023]. However, the refiners are trained for each new domain. Finally, Yang et al. [2022] use prompted feedback and refinement specifically tailored for story generation. In this work, we avoid training a separate refiner, and show that the same model can be used as both the refiner and the source of feedback across multiple domains.

**Non-refinement reinforcement learning (RL) approaches.** Rather than having explicit refinement, an alternative way to incorporate feedback is by optimizing a scalar reward function, e.g. with reinforcement learning (e.g., Stiennon et al. [2020], Lu et al. [2022], Le et al. [2022a]). These methods differ from SELF-REFINE in that the model does not access feedback on an intermediate generation. Second, these RL methods require updating the model's parameters, unlike SELF-REFINE. Recently, in discrete-space simulated environments, LLMs have also been shown to iteratively shape and refine rewards and policies, thereby performing RL tasks without expert demonstrations or gradients [Kim et al., 2023, Brooks et al., 2023]. While we focus on real-world code and language tasks in this paper, it would be interesting to explore applications of self-refine in simulated environments.

## 10.6 Limitations and Discussion

The main limitation of our approach is that the base models need to have sufficient few-shot modeling or instruction-following abilities, in order to learn to provide feedback and to refine in an in-context fashion, without having to train supervised models and rely on supervised data.

Further, the experiments in this work were primarily performed with language models that are not open-sourced, namely GPT-3, GPT-3.5, GPT-4, and CODEX. Existing literature [Ouyang et al., 2022a] does not fully describe the details of these models, such as the pretraining corpus, model sizes, and model biases. Nonetheless, we release our code and model outputs to ensure the reproducibility of our work. In addition, initial results from our experiments with the open-access LLAMA-2-70B language model are promising, reinforcing the notion that SELF-REFINE has the potential to be widely applicable, even as open-source models continue to evolve and improve.

Another limitation of our work is that we exclusively experiment with datasets in English. In other languages, the current models may not provide the same benefits. Finally, there is a possibility for bad actors to use prompting techniques to steer a model to generate more toxic or harmful text. Our approach does not explicitly guard against this.

## 10.7 Conclusion

We present SELF-REFINE: a novel approach that allows large language models to iteratively provide self-feedback and refine their own outputs. SELF-REFINE operates within a single LLM, requiring neither additional training data nor reinforcement learning. We demonstrate the simplicity and ease of use of SELF-REFINE across a wide variety of tasks. By showcasing the potential of SELF-REFINE in diverse tasks, our research contributes to the ongoing exploration and development of large language models, with the aim of reducing the cost of human creative processes in real-world settings. We hope that our iterative approach will help drive further research in this area. To this end, we make all our code, data and prompts available at https://selfrefine.info/.

# Part V

# A Case for Inference-time Compute: Conclusion and Future Work

Training large language models requires tremendous engineering efforts spanning several sub-fields of computer science. However, the key idea is surprisingly simple: Train a $\mathcal{O}$(billions) parameter transformer-based language model to predict the next token (autoregressive generation) with a corpus of $\mathcal{O}$(trillion) tokens. This recipe has yielded language models (LMs) that can outperform humans in programming challenges [Li et al., 2021b, Shypula et al., 2023, Ridnik et al., 2024], contribute to novel discoveries [Romera-Paredes et al., 2024, Trinh et al., 2024], and demonstrate high proficiency in complex reasoning tasks [Wu et al., 2022a, Suzgun et al., 2022b, Zhao et al., 2023, Zhang et al., 2024].

Proponents of LMs argue that while next-token prediction may seem straightforward, learning to do it reliably is non-trivial. Just as humans learn to draw on an extensive, nuanced understanding of the world [Bennett, 2023], LMs trained on vast amounts of data are emulating a similar process [Li et al., 2021a, 2022, Gurnee and Tegmark, 2023], with the advantage of being exposed to orders of magnitude more information than a single human can process in a lifetime [Brown et al., 2020c, Chowdhery et al., 2022b, Touvron et al., 2023]. However, recent research has highlighted the limitations of autoregressive generation in complex reasoning [Madaan et al., 2023b, LeCun, 2023], compositionality [Dziri et al., 2024], and generalization [Berglund et al., 2023], revealing a significant gap between the capabilities and styles of humans and LMs.

These seemingly contradictory facts—LMs excelling in some areas while failing in others—have led to two divergent camps within the AI community. One camp emphasizes LMs' potential to solve complex problems, with some suggesting the possibility of matching (Artificial General Intelligence or AGI [Goertzel and Pennachin, 2007, Morris et al., 2023]) or greatly surpassing (Artificial Super Intelligence or ASI [Bostrom, 2014, Shanahan, 2015]) human intelligence across all practical domains.[3]. On the other hand, critics contend that LMs merely recombine the data they have been trained on.

Between these two extremes, a middle ground emerges: utilizing LMs as a central component within a larger reasoning system and developing complementary data preprocessing, modeling, training, and post-inference techniques. An analogy is drawing samples from a more complex distribution by starting with a simple uniform distribution and applying a transformation. Similarly, LMs can be seen as a starting point, capturing a snapshot of human knowledge. We can then apply various techniques to transform and refine the LLM's output, allowing us to solve more complex problems.

The works presented in this thesis occupy this middle ground, with language models playing

[3]While the idea of AGI has been revived due to the current stream of LMs, the idea that machines can one day be as smart or even more intelligent than humans have been part of the narrative since the early days of modern computing. See Waldrop [2018] for an overview of the early days of AI and discussions on superhuman intelligence.

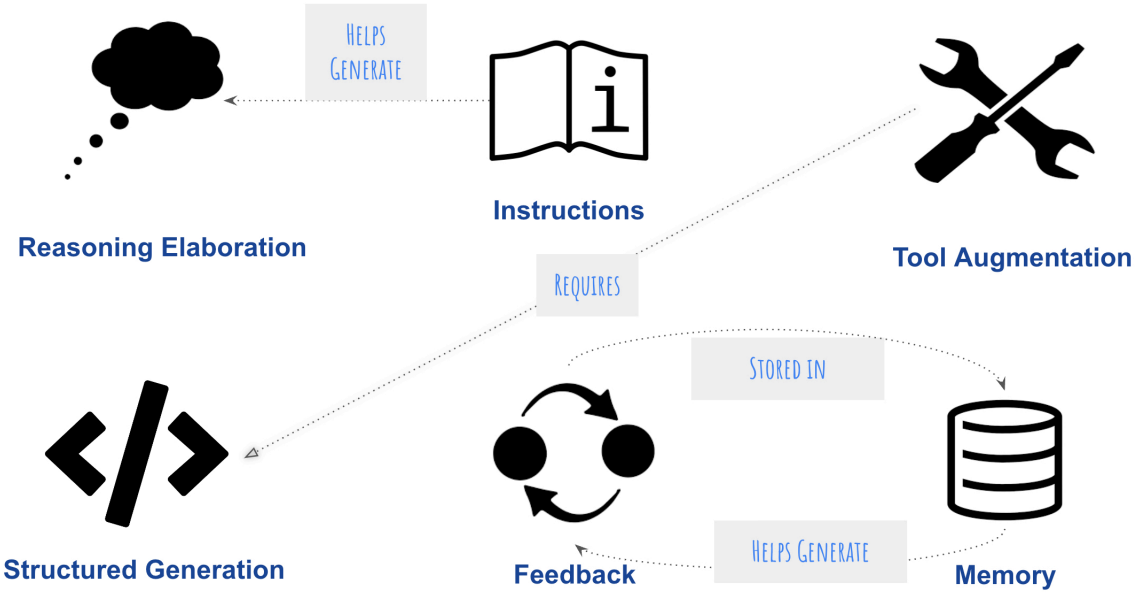



Figure 10.6: The next generation of large language models will adopt a systems perspective, with the LLM serving as a central proposal distribution augmented by inference-time computation techniques like reasoning elaboration, structured generation, incorporation of memory and feedback, and integration with external tools - enabling more robust and versatile reasoning capabilities compared to raw LLM alone.

a central role and various techniques (e.g.,, code-based prompting or self-refinement) being developed to harness their strengths. To continue the analogy, the language model serves as a prior distribution, and the proposed techniques transform the prior to enable drawing good samples.

We posit that large language models acting as kernels, upon which rich inference procedures must be built, will be central to advances in complex reasoning. This will involve building techniques like search algorithms and more expressive high-level languages to leverage the LM's ability to act as a prior distribution. This idea, which we call *inference-time compute*, is one direction from which the next set of breakthroughs might come.

### 10.7.1 The Necessity of Inference-Time Computation for Complex Reasoning

Just as humans dedicate more thought and effort to complex problems [Posner and Snyder, 1975, Shiffrin and Schneider, 1977, Evans, 1984, Stanovich, 2000, Kahneman, 2003, Frankish, 2010, Kahneman, 2011], we argue that the next generation of language models should be able to allocate more computational resources to tackle harder tasks dynamically. However, current language models rely on a single forward pass to handle both the simplest and the most complex queries. As we discussed in Chapter 1, this is unnatural and incompatible with most everyday creative tasks.

We posit that the next generation of AI systems will infuse the idea of leveraging more compute

to solve the more challenging problems. The idea of adaptive computation is not new. Broadly, at least three different lines of effort have been proposed to leverage adaptive computation:

**Option 1: Training Larger Models** The remarkable success of scaling laws has fueled a focus on training increasingly large models [Hoffmann et al., 2022, Aghajanyan et al., 2023, Kaplan et al., 2020]. These laws suggest that language model performance improves predictably with increases in the number of parameters and dataset size. There is strong evidence this trend will continue (e.g., GPT-4 [OpenAI, 2023] and Gemini [Team et al., 2023]), yielding models capable of even more complex reasoning. However, the sheer scale of these models poses challenges. Training and running them requires massive computational resources due to the power-law relationship between compute and performance gains. Additionally, their inference speed can be a bottleneck in real-world applications. While scaling laws offer potential, these challenges highlight the need for complementary techniques.

**Option 2: Using Architectures that Use Adaptive Computation** This approach aims to embed flexibility directly into model architectures, allowing them to adjust their computational effort dynamically. Techniques in this class include early stopping (halting training or inference when performance plateaus [Liu et al., 2020, Zhou et al., 2020, Schuster et al., 2021, Geng et al., 2021]), Universal Transformers (which adapt the depth of the network based on input complexity [Dehghani et al., 2019]), and Memorizing Transformers (which are trained to attend to additional variable length states retrieved using KNNs, [Wu et al., 2022b]). The promise is greater efficiency and the potential to tackle more complex tasks. However, these architectures can be notoriously difficult to train and scale effectively, a possible example of the bitter lesson in AI [Sutton, 2019], where complex solutions often fall short compared to simpler, scalable approaches.

**Option 3: Using Inference-Time Compute** This strategy positions a powerful language model as a central component and leverages additional computational procedures during inference. This could involve techniques like knowledge retrieval [Liu et al., 2022a], task decomposition [Zhou et al., 2022], eliciting reasoning process from the model [Wei et al., 2022b], or integration with external tools [Gao et al., 2023]. This approach aligns most closely with the techniques explored in this thesis.

Next, we will outline some techniques that fall in the category of inference-time compute, and some promising avenues for more progress.

## 10.8 Enhancing LLM Problem-Solving with Inference-Time Computation

While the concept of inference-time computation is still evolving, several techniques have already demonstrated remarkable success in enhancing language model capabilities. This section outlines established methods with proven results alongside promising directions that could pave the way

for the next generation of reasoning systems. We will explore how these techniques transform LMs into more robust and adaptable reasoning systems.

### 10.8.1 Search

**Rejection Sampling** Rejection sampling is a technique that allows us to draw higher-quality samples from a distribution defined by a language model. In the case of AlphaCode [Li et al., 2021b], rejection sampling was used to generate multiple candidate code solutions and filter out those that failed to meet specific criteria. This approach demonstrably improved the quality of generated solutions, showcasing the power of targeted inference-time computation.

**Planning and Tree-Search** For complex problems, the space of potential solutions can be enormous. Exhaustive search is impractical, while rejection sampling can be wasteful. Planning techniques like Monte-Carlo Tree Search (MCTS) offer a powerful solution [Coulom, 2006]. MCTS intelligently allocates computational resources by focusing exploration on the most promising areas of the solution space, and has been famously applied for systems like AlphaGo [Silver et al., 2016, 2017].

### 10.8.2 Context-sensitive inference

Currently, each forward pass is "context-free." The mistakes are repeated, and the feedback is wasted. This starkly contrasts with humans, where learning and inference work in tandem.

This context-free nature of traditional language model inference limits their ability to learn from past mistakes, incorporate feedback, and adapt to evolving situations. To address this, we must develop context-sensitive inference procedures. Two promising techniques are:

**Memory** By using a memory, LMs could recall previous outputs and interactions [Madaan et al., 2022b, Zhang et al., 2024]. For instance, this technique can prevent the model from making the same mistake twice, making interactions more efficient and enabling improvements without retraining.

**Lightweight Model Editing** Methods to directly update model parameters based on feedback would allow LMs to learn from their errors and improve performance over time [Example: A user corrects factual inaccuracy, and the LLM adjusts its knowledge base accordingly].

**Feedback** Unlike humans, LMs traditionally stop learning once deployed. This stark contrast limits their ability to adapt and improve over time. The integration of feedback mechanisms has the potential to change this fundamentally. While memprompt [Tandon et al., 2022a, Madaan et al., 2022b] takes the first steps towards this goal, it is important to remember that some related techniques, such as effective retrieval and reasoning over long-contexts, will likely be important as well.

### 10.8.3 Agents and Multimodality

The idea of composing systems from specialized language models (e.g., AutoMix [Madaan et al., 2023a]) offers a flexible and powerful approach to complex reasoning. As inference costs decrease, this paradigm will likely gain even more traction. We will see a growing need for models that consume diverse modalities like images, videos, and code [Zhou et al., 2023c]. Importantly, these specialties may be supported by different language model architectures, creating a dynamic "network" of neural networks. This approach allows us to leverage the complementary strengths of different models and modalities for more robust problem-solving.

# Bibliography


Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *An exact graph edit distance algorithm for solving pattern recognition problems*, 2015.

Armen Aghajanyan, Lili Yu, Alexis Conneau, Wei-Ning Hsu, Karen Hambardzumyan, Susan Zhang, Stephen Roller, Naman Goyal, Omer Levy, and Luke Zettlemoyer. Scaling laws for generative mixed-modal language models. *ArXiv preprint*, abs/2301.03728, 2023. URL https://arxiv.org/abs/2301.03728.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as I Can, not as I Say: Grounding Language in Robotic Affordances. *ArXiv preprint*, abs/2204.01691, 2022. URL https://arxiv.org/abs/2204.01691.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-wesley Reading, 2007.

James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

Teresa M. Amabile. A Theoretical Framework. In *The Social Psychology of Creativity*, pages 65–96. Springer New York, New York, NY, 1983. ISBN 978-1-4612-5533-8. doi: 10.1007/978-1-4612-5533-8_4. URL https://doi.org/10.1007/978-1-4612-5533-8_4.

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. In *ACL*, 2019.

Daniel Andor, Luheng He, Kenton Lee, and Emily Pitler. Giving BERT a calculator: Finding operations and arguments with reading comprehension. In *Proc. of EMNLP*, pages 5947–5952, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1609. URL https://aclanthology.org/D19-1609.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David

Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *ArXiv preprint*, abs/1607.06450, 2016. URL https://arxiv.org/abs/1607.06450.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *Proc. of ICLR*, 2015a. URL http://arxiv.org/abs/1409.0473.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *Proc. of ICLR*, 2015b. URL http://arxiv.org/abs/1409.0473.

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron C. Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. In *Proc. of ICLR*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJDaqqveg.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *ArXiv preprint*, abs/2204.05862, 2022a. URL https://arxiv.org/abs/2204.05862.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *ArXiv preprint*, abs/2212.08073, 2022b. URL https://arxiv.org/abs/2212.08073.

Jasmijn Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. Graph convolutional encoders for syntax-aware neural machine translation. In *Proc. of EMNLP*, pages 1957–1967, Copenhagen, Denmark, 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1209. URL https://aclanthology.org/D17-1209.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *ArXiv preprint*, abs/1806.01261, 2018. URL https://arxiv.org/abs/1806.01261.

Emily M. Bender and Alexander Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proc. of ACL*, pages 5185–5198, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.463. URL https://aclanthology.org/2020.acl-main.463.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8): 1798–1828, 2013.

Max Bennett. *A Brief History of Intelligence: Evolution, AI, and the Five Breakthroughs That Made Our Brains*. Mariner Books, New York, 2023. ISBN 9780063286344.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proc. of EMNLP*, pages 1533–1544, Seattle, Washington, USA, 2013. Association for Computational Linguistics. URL https://aclanthology.org/D13-1160.

Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating Python Performance Issues with SCALENE. *ArXiv preprint*, abs/2212.07597, 2022. URL https://arxiv.org/abs/2212.07597.

Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *ArXiv preprint*, abs/2309.12288, 2023. URL https://arxiv.org/abs/2309.12288.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.

Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Celikyilmaz, and Yejin Choi. COMET: Commonsense transformers for automatic knowledge graph construction. In *Proc. of ACL*, pages 4762–4779, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1470. URL https://aclanthology.org/P19-1470.

Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, Oxford, 2014.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proc. of EMNLP*, pages 632–642, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1075. URL https://aclanthology.org/D15-1075.

Philip Bramsen, Martha Escobar-Molano, Ami Patel, and Rafael Alonso. Extracting social power relationships from natural language. In *Proc. of ACL*, pages 773–782, Portland, Oregon, USA, 2011. Association for Computational Linguistics. URL https://aclanthology.org/P11-1078.

Ethan Brooks, Logan Walls, Richard L. Lewis, and Satinder Singh. Large language models can implement policy iteration, 2023.

Lawrence D Brown, T Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statistical science*, 16(2):101–133, 2001.

Penelope Brown, Stephen C Levinson, and Stephen C Levinson. *Politeness: Some universals in language usage*, volume 4. Cambridge university press, 1987.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In

Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020a. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020b. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020c. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Paweł Budzianowski and Ivan Vulić. Hello, it's GPT-2 - how can I help you? towards the use of pretrained language models for task-oriented dialogue systems. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 15–22, Hong Kong, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5602. URL https://aclanthology.org/D19-5602.

Yuri Burda, Roger B. Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. In Yoshua Bengio and Yann LeCun, editors, *Proc. of ICLR*, 2016. URL http://arxiv.org/abs/1509.00519.

Meng Cao, Yue Dong, and Jackie Cheung. Hallucinated but factual! inspecting the factuality of hallucinations in abstractive summarization. In *Proc. of ACL*, pages 3340–3354, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.236. URL https://aclanthology.org/2022.acl-long.236.

Taylor Cassidy, Bill McDowell, Nathanael Chambers, and Steven Bethard. An annotation framework for dense event ordering. In *Proc. of ACL*, pages 501–506, Baltimore, Maryland, 2014. Association for Computational Linguistics. doi: 10.3115/v1/P14-2082. URL https:

//aclanthology.org/P14-2082.

Nathanael Chambers and Dan Jurafsky. Unsupervised learning of narrative schemas and their participants. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 602–610, Suntec, Singapore, 2009. Association for Computational Linguistics. URL https://aclanthology.org/P09-1068.

Nathanael Chambers, Taylor Cassidy, Bill McDowell, and Steven Bethard. Dense event ordering with a multi-pass architecture. *Transactions of the Association for Computational Linguistics*, 2:273–284, 2014. doi: 10.1162/tacl_a_00182. URL https://aclanthology.org/Q14-1022.

Khyathi Chandu, Shrimai Prabhumoye, Ruslan Salakhutdinov, and Alan W Black. "my way of telling a story": Persona based grounded story generation. In *Proceedings of the Second Workshop on Storytelling*, pages 11–21, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-3402. URL https://aclanthology.org/W19-3402.

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, et al. AlpaGasus: Training A Better Alpaca with Fewer Data. *ArXiv preprint*, abs/2307.08701, 2023. URL https://arxiv.org/abs/2307.08701.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision Transformer: Reinforcement Learning via Sequence Modeling. *NeurIPS*, 2021a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *ArXiv preprint*, abs/2107.03374, 2021b. URL https://arxiv.org/abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie

Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *ArXiv preprint*, abs/2107.03374, 2021c. URL https://arxiv.org/abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021d. URL https://arxiv.org/abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *ArXiv preprint*, abs/2211.12588, 2022. URL https://arxiv.org/abs/2211.12588.

Xiaohui Chen, Xu Han, Jiajing Hu, Francisco J. R. Ruiz, and Li-Ping Liu. Order matters: Probabilistic modeling of node sequence for graph generation. In Marina Meila and Tong Zhang, editors, *Proc. of ICML*, volume 139 of *Proceedings of Machine Learning Research*, pages 1630–1639. PMLR, 2021e. URL http://proceedings.mlr.press/v139/chen21j.html.

Xilun Chen, Ahmed Hassan Awadallah, Hany Hassan, Wei Wang, and Claire Cardie. Multi-source cross-lingual model transfer: Learning what to share. In *Proc. of ACL*, pages 3098–3112, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1299. URL https://aclanthology.org/P19-1299.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. *ArXiv preprint*, abs/2210.02875, 2022. URL https://arxiv.org/abs/2210.02875.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. Vicuna: An open-source chatbot impressing gpt-4 with 90

quality, 2023.

Eunsol Choi, Omer Levy, Yejin Choi, and Luke Zettlemoyer. Ultra-fine entity typing. In *Proc. of ACL*, pages 87–96, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1009. URL https://aclanthology.org/P18-1009.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason

Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. *ArXiv preprint*, abs/2204.02311, 2022a. URL https://arxiv.org/abs/2204.02311.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *ArXiv preprint*, abs/2204.02311, 2022b. URL https://arxiv.org/abs/2204.02311.

Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv preprint*, abs/2110.14168, 2021. URL https://arxiv.org/abs/2110.14168.

Liz Coppock. Politeness strategies in conversation closings, 2005.

Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

Hongliang Dai, Yangqiu Song, and Haixun Wang. Ultra-fine entity typing with weak supervision from a masked language model. In *Proc. of ACL*, pages 1790–1799, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.141. URL https://aclanthology.org/2021.acl-long.141.

Bhavana Dalvi, Lifu Huang, Niket Tandon, Wen-tau Yih, and Peter Clark. Tracking state changes in procedural text: a challenge dataset and models for process paragraph comprehension. In *Proc. of NAACL-HLT*, pages 1595–1604, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1144. URL https://aclanthology.org/N18-1144.

Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. A computational approach to politeness with application to social factors. In *Proc. of ACL*, pages 250–259, Sofia, Bulgaria, 2013. Association for Computational Linguistics. URL https://aclanthology.org/P13-1025.

Sanjoy Dasgupta, Daniel Hsu, Stefanos Poulis, and Xiaojin Zhu. Teaching a black-box learner. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proc. of ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 1547–1555. PMLR, 2019. URL http://proceedings.mlr.press/v97/dasgupta19a.html.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *Proc. of ICLR*. OpenReview.net, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

David Demeter and Doug Downey. Just add functions: A neural-symbolic language model. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second*

*Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 7634–7642. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/6264.

Dorottya Demszky, Dana Movshovitz-Attias, Jeongwoo Ko, Alan Cowen, Gaurav Nemade, and Sujith Ravi. GoEmotions: A dataset of fine-grained emotions. In *Proc. of ACL*, pages 4040–4054, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.372. URL https://aclanthology.org/2020.acl-main.372.

Michael Denkowski and Alon Lavie. Meteor 1.3: Automatic metric for reliable optimization and evaluation of machine translation systems. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 85–91, Edinburgh, Scotland, 2011. Association for Computational Linguistics. URL https://aclanthology.org/W11-2107.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. of NAACL-HLT*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. Language model cascades. *ArXiv preprint*, abs/2207.10342, 2022. URL https://arxiv.org/abs/2207.10342.

Justin Domke and Daniel R. Sheldon. Importance weighting and variational inference. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4475–4484, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/25db67c5657914454081c6a18e93d6dd-Abstract.html.

Rotem Dror, Gili Baumer, Segev Shlomov, and Roi Reichart. The hitchhiker's guide to testing statistical significance in natural language processing. In *Proc. of ACL*, pages 1383–1392, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1128. URL https://aclanthology.org/P18-1128.

Wanyu Du, Zae Myung Kim, Vipul Raheja, Dhruv Kumar, and Dongyeop Kang. Read, revise, repeat: A system demonstration for human-in-the-loop iterative text revision. In *Proceedings of the First Workshop on Intelligent and Interactive Writing Assistants (In2Writing 2022)*, pages 96–108, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.in2writing-1.14. URL https://aclanthology.org/2022.in2writing-1.14.

Nicholas D Duran, Philip M McCarthy, Art C Graesser, and Danielle S McNamara. Using temporal cohesion to predict temporal coherence in narrative and expository texts. *Behavior Research Methods*, 39(2):212–223, 2007.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. In *Proc. of NAACL-HLT*, pages 199–209, San Diego, California, 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1024. URL https://aclanthology.

org/N16-1024.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36, 2024.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online, 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.444. URL https://aclanthology.org/2021.naacl-main.444.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online, 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.444. URL https://aclanthology.org/2021.naacl-main.444.

Jonathan St B. T. Evans. Heuristic and analytic processes in reasoning*. *British Journal of Psychology*, 75(4):451–468, 1984. ISSN 2044-8295. doi: 10.1111/j.2044-8295.1984.tb01915.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8295.1984.tb01915.x. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2044-8295.1984.tb01915.x.

Jacqueline Evers-Vermeul, Jet Hoek, and Merel CJ Scholman. On temporality in discourse annotation: Theoretical and practical considerations. *Dialogue and Discourse*, 8(2):1–20, 2017.

et al. Falcon, WA. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *ArXiv preprint*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961.

Linda Flower and John R Hayes. A cognitive process theory of writing. *College composition and communication*, 32(4):365–387, 1981.

Maxwell Forbes, Jena D. Hwang, Vered Shwartz, Maarten Sap, and Yejin Choi. Social chemistry 101: Learning to reason about social and moral norms. In *Proc. of EMNLP*, pages 653–670, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.48. URL https://aclanthology.org/2020.emnlp-main.48.

Keith Frankish. Dual-Process and Dual-System Theories of Reasoning. *Philosophy Compass*, 5(10):914–926, 2010. ISSN 1747-9991. doi: 10.1111/j.1747-9991.2010.00330.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1747-9991.2010.00330.x. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1747-9991.2010.00330.x.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong,

Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *ArXiv preprint*, abs/2204.05999, 2022a. URL https://arxiv.org/abs/2204.05999.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A Generative Model for Code Infilling and Synthesis. *ArXiv preprint*, abs/2204.05999, 2022b. URL https://arxiv.org/abs/2204.05999.

Hao Fu, Yao; Peng and Tushar Khot. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*, 2022. URL https://shorturl.at/ouI79.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. Gptscore: Evaluate as you desire. *ArXiv preprint*, abs/2302.04166, 2023. URL https://arxiv.org/abs/2302.04166.

Zhenxin Fu, Xiaoye Tan, Nanyun Peng, Dongyan Zhao, and Rui Yan. Style transfer in text: Exploration and evaluation. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proc. of AAAI*, pages 663–670. AAAI Press, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17015.

Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. A theoretical analysis of the repetition problem in text generation. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 12848–12856. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/17520.

Chuang Gan, Zhe Gan, Xiaodong He, Jianfeng Gao, and Li Deng. Stylenet: Generating attractive visual captions with styles. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 955–964. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.108. URL https://doi.org/10.1109/CVPR.2017.108.

Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2006.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: the 3rd wave. *ArXiv preprint*, abs/2012.05876, 2020. URL https://arxiv.org/abs/2012.05876.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. DeepPERF: A Deep Learning-Based Approach For Improving Software Performance, 2022. URL https://arxiv.org/abs/2206.13619.

Sebastian Gehrmann, Tosin Adewumi, Karmanya Aggarwal, Pawan Sasanka Ammanamanchi, Aremu Anuoluwapo, Antoine Bosselut, Khyathi Raghavi Chandu, Miruna Clinciu, Dipanjan Das, Kaustubh D. Dhole, Wanyu Du, Esin Durmus, Ondřej Dušek, Chris Emezue, Varun Gangal, Cristina Garbacea, Tatsunori Hashimoto, Yufang Hou, Yacine Jernite, Harsh Jhamtani, Yangfeng

Ji, Shailza Jolly, Mihir Kale, Dhruv Kumar, Faisal Ladhak, Aman Madaan, Mounica Maddela, Khyati Mahajan, Saad Mahamood, Bodhisattwa Prasad Majumder, Pedro Henrique Martins, Angelina McMillan-Major, Simon Mille, Emiel van Miltenburg, Moin Nadeem, Shashi Narayan, Vitaly Nikolaev, Rubungo Andre Niyongabo, Salomey Osei, Ankur Parikh, Laura Perez-Beltrachini, Niranjan Ramesh Rao, Vikas Raunak, Juan Diego Rodriguez, Sashank Santhanam, João Sedoc, Thibault Sellam, Samira Shaikh, Anastasia Shimorina, Marco Antonio Sobrevilla Cabezudo, Hendrik Strobelt, Nishant Subramani, Wei Xu, Diyi Yang, Akhila Yerukola, and Jiawei Zhou. The GEM Benchmark: Natural Language Generation, its Evaluation and Metrics. *ArXiv preprint*, abs/2102.01672, 2021a. URL https://arxiv.org/abs/2102.01672.

Sebastian Gehrmann, Tosin Adewumi, Karmanya Aggarwal, Pawan Sasanka Ammanamanchi, Anuoluwapo Aremu, Antoine Bosselut, Khyathi Raghavi Chandu, Miruna-Adriana Clinciu, Dipanjan Das, Kaustubh Dhole, Wanyu Du, Esin Durmus, Ondřej Dušek, Chris Chinenye Emezue, Varun Gangal, Cristina Garbacea, Tatsunori Hashimoto, Yufang Hou, Yacine Jernite, Harsh Jhamtani, Yangfeng Ji, Shailza Jolly, Mihir Kale, Dhruv Kumar, Faisal Ladhak, Aman Madaan, Mounica Maddela, Khyati Mahajan, Saad Mahamood, Bodhisattwa Prasad Majumder, Pedro Henrique Martins, Angelina McMillan-Major, Simon Mille, Emiel van Miltenburg, Moin Nadeem, Shashi Narayan, Vitaly Nikolaev, Andre Niyongabo Rubungo, Salomey Osei, Ankur Parikh, Laura Perez-Beltrachini, Niranjan Ramesh Rao, Vikas Raunak, Juan Diego Rodriguez, Sashank Santhanam, João Sedoc, Thibault Sellam, Samira Shaikh, Anastasia Shimorina, Marco Antonio Sobrevilla Cabezudo, Hendrik Strobelt, Nishant Subramani, Wei Xu, Diyi Yang, Akhila Yerukola, and Jiawei Zhou. The GEM benchmark: Natural language generation, its evaluation and metrics. In *Proceedings of the 1st Workshop on Natural Language Generation, Evaluation, and Metrics (GEM 2021)*, pages 96–120, Online, 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.gem-1.10. URL https://aclanthology.org/2021.gem-1.10.

Edward M Gellenbeck and Curtis R Cook. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*, pages 65–81. Ablex Publishing, Norwood, NJ, 1991.

Shijie Geng, Peng Gao, Zuohui Fu, and Yongfeng Zhang. Romebert: Robust training of multi-exit bert. *ArXiv preprint*, abs/2101.09755, 2021. URL https://arxiv.org/abs/2101.09755.

Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song. Koala: A dialogue model for academic research. Blog post, 2023. URL https://bair.berkeley.edu/blog/2023/04/03/koala/.

Mor Geva, Ankit Gupta, and Jonathan Berant. Injecting numerical reasoning skills into language models. In *Proc. of ACL*, pages 946–958, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.89. URL https://aclanthology.org/2020.acl-main.89.

S. Ghosh, Giedrius Burachas, Arijit Ray, and Avi Ziskind. Generating natural language explanations for visual question answering using scene graphs and visual attention. *ArXiv preprint*, abs/1902.05715, 2019. URL https://arxiv.org/abs/1902.05715.

Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*, volume 2. Springer, 2007.

Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

Colin Graber, Ofer Meshi, and Alexander G. Schwing. Deep structured prediction with nonlinear output transformations. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6323–6334, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/a2d10d355cdebc879e4fc6ecc6f63dd7-Abstract.html.

Jiatao Gu, Hany Hassan, Jacob Devlin, and Victor O.K. Li. Universal neural machine translation for extremely low resource languages. In *Proc. of NAACL-HLT*, pages 344–354, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1032. URL https://aclanthology.org/N18-1032.

Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. Neural module networks for reasoning over text. In *Proc. of ICLR*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SygWvAVFPr.

Prakhar Gupta, Cathy Jiao, Yi-Ting Yeh, Shikib Mehri, Maxine Eskenazi, and Jeffrey P Bigham. InstructDial: Improving Zero and Few-shot Generalization in Dialogue through Instruction Tuning. In *EMNLP*, 2022.

Wes Gurnee and Max Tegmark. Language models represent space and time. *ArXiv preprint*, abs/2310.02207, 2023. URL https://arxiv.org/abs/2310.02207.

Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008a.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008b.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language Models can Teach Themselves to Program Better. *ArXiv preprint*, abs/2207.14502, 2022. URL https://arxiv.org/abs/2207.14502.

Xu Han, Tianyu Gao, Yankai Lin, Hao Peng, Yaoliang Yang, Chaojun Xiao, Zhiyuan Liu, Peng Li, Jie Zhou, and Maosong Sun. More data, more relations, more context and more openness: A review and outlook for relation extraction. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 745–758, Suzhou, China, 2020. Association for Computational Linguistics. URL https://aclanthology.org/2020.aacl-main.75.

Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards

a Unified View of Parameter-Efficient Transfer Learning. *ArXiv preprint*, abs/2110.04366, 2021. URL https://arxiv.org/abs/2110.04366.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Ruining He and Julian J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517. ACM, 2016. doi: 10.1145/2872427.2883037. URL https://doi.org/10.1145/2872427.2883037.

Micha Heilbron, Kristijan Armeni, Jan-Mathijs Schoffelen, Peter Hagoort, and Floris P De Lange. A hierarchy of linguistic predictions during natural language comprehension. *Proceedings of the National Academy of Sciences*, 119(32):e2201968119, 2022.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset, 2021. URL https://openreview.net/forum?id=7Bywt2mQsCe.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *ArXiv preprint*, abs/2203.15556, 2022. URL https://arxiv.org/abs/2203.15556.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *Proc. of ICLR*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

Eduard Hovy, Roberto Navigli, and Simone Paolo Ponzetto. Collaboratively built semi-structured content and artificial intelligence: The story so far. *Artificial Intelligence*, 194:2–27, 2013.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *ArXiv preprint*, abs/2106.09685, 2021. URL https://arxiv.org/abs/2106.09685.

Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P. Xing. Toward controlled generation of text. In Doina Precup and Yee Whye Teh, editors, *Proc. of ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 1587–1596. PMLR, 2017. URL http://proceedings.mlr.press/v70/hu17e.html.

Jena D. Hwang, Chandra Bhagavatula, Ronan Le Bras, Jeff Da, Keisuke Sakaguchi, Antoine Bosselut, and Yejin Choi. (comet-) atomic 2020: On symbolic and neural commonsense

knowledge graphs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6384–6392. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16792.

Daniel Jiwoong Im, Sungjin Ahn, Roland Memisevic, and Yoshua Bengio. Denoising criterion for variational auto-encoding framework. In Satinder P. Singh and Shaul Markovitch, editors, *Proc. of AAAI*, pages 2059–2065. AAAI Press, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14213.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Larry L. Jacoby and Christopher N. Wahlheim. On the importance of looking back: The role of recursive remindings in recency judgments and cued recall. *Memory and Cognition*, 41: 625–637, 2013.

Natasha Jaques, Asma Ghandeharioun, Judy Hanwen Shen, Craig Ferguson, Agata Lapedriza, Noah Jones, Shixiang Gu, and Rosalind Picard. Way off-policy batch deep reinforcement learning of implicit human preferences in dialog. *ArXiv preprint*, abs/1907.00456, 2019. URL https://arxiv.org/abs/1907.00456.

Harsh Jhamtani, Varun Gangal, Eduard Hovy, and Eric Nyberg. Shakespearizing modern language using copy-enriched sequence to sequence models. In *Proceedings of the Workshop on Stylistic Variation*, pages 10–19, Copenhagen, Denmark, 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-4902. URL https://aclanthology.org/W17-4902.

Liwei Jiang, Jena D Hwang, Chandra Bhagavatula, Ronan Le Bras, Maxwell Forbes, Jon Borchardt, Jenny Liang, Oren Etzioni, Maarten Sap, and Yejin Choi. Delphi: Towards machine ethics and norms. *ArXiv preprint*, abs/2110.07574, 2021. URL https://arxiv.org/abs/2110.07574.

Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In Jennifer G. Dy and Andreas Krause, editors, *Proc. of ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 2328–2337. PMLR, 2018. URL http://proceedings.mlr.press/v80/jin18a.html.

Vineet John, Lili Mou, Hareesh Bahuleyan, and Olga Vechtomova. Disentangled representation learning for non-parallel text style transfer. In *Proc. of ACL*, pages 424–434, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1041. URL https://aclanthology.org/P19-1041.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 2019a.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019b.

Richard A Johnson, Irwin Miller, and John E Freund. *Probability and statistics for engineers*, volume 2000. Pearson Education London, 2000.

P. Johnson-Laird. *Mental Models : Towards a Cognitive Science of Language*. Harvard University Press, 1983.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

Michael I Jordan and Lei Xu. Convergence results for the em approach to mixtures of experts architectures. *Neural networks*, 8(9):1409–1431, 1995.

Xincheng Ju, Dong Zhang, Junhui Li, and Guodong Zhou. Transformer-based label set generation for multi-modal multi-label emotion detection. In *Proceedings of the 28th ACM international conference on multimedia*, pages 512–520, 2020.

Daniel Jurafsky, Elizabeth Shriberg, and Debra Biasca. Switchboard SWBD-DAMSL shallow-discourse-function annotation coders manual, draft 13. Technical Report 97-02, University of Colorado, Boulder Institute of Cognitive Science, Boulder, CO, 1997.

Dániel Z Kádár and Sara Mills. *Politeness in East Asia*. Cambridge University Press, 2011.

Daniel Kahneman. Maps of Bounded Rationality: Psychology for Behavioral Economics. *The American Economic Review*, 93(5):1449–1475, 2003. ISSN 0002-8282. URL https://www.jstor.org/stable/3132137. Publisher: American Economic Association.

Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.

Dongyeop Kang and Eduard Hovy. xslue: A benchmark and analysis platform for cross-style language understanding and evaluation, 2019.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *ArXiv preprint*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 21:23, 2015.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *ArXiv preprint*, abs/2303.17491, 2023. URL https://arxiv.org/abs/2303.17491.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proc. of ICLR*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. MAWPS: A math word problem repository. In *Proc. of NAACL-HLT*, pages 1152–1157, San Diego, California, 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1136. URL https://aclanthology.org/N16-1136.

Robert Koons. Defeasible Reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2017 edition, 2017.

Adam R Kosiorek, Hyunjik Kim, and Danilo J Rezende. Conditional set generation with transformers. *ArXiv preprint*, abs/2006.16841, 2020. URL https://arxiv.org/abs/2006.16841.

Julia Kreutzer, Joshua Uyheng, and Stefan Riezler. Reliability and learnability of human bandit feedback for sequence-to-sequence reinforcement learning. In *Proc. of ACL*, pages 1777–1788, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1165. URL https://aclanthology.org/P18-1165.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *ArXiv preprint*, abs/2207.01780, 2022a. URL https://arxiv.org/abs/2207.01780.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv preprint*, abs/2207.01780, 2022b. URL https://arxiv.org/abs/2207.01780.

Y LeCun. Do large language models need sensory grounding for meaning and understanding? In *Workshop on Philosophy of Deep Learning, NYU Center for Mind, Brain, and Consciousness and the Columbia Center for Science and Society*, 2023.

David Lewis. Reuters-21578 text categorization test collection, distribution 1.0. *http://www.research.att.com*, 1997.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proc. of ACL*, pages 7871–7880, Online, 2020a. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL https://aclanthology.org/2020.acl-main.703.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020b. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. *ArXiv preprint*, abs/2206.14858, 2022. URL https://arxiv.org/abs/2206.14858.

Belinda Z. Li, Maxwell Nye, and Jacob Andreas. Implicit representations of meaning in neural language models. In *Proc. of ACL*, pages 1813–1827, Online, 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.143. URL https://aclanthology.org/2021.acl-long.143.

Juncen Li, Robin Jia, He He, and Percy Liang. Delete, retrieve, generate: a simple approach to sentiment and style transfer. In *Proc. of NAACL-HLT*, pages 1865–1874, New Orleans,

Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1169. URL https://aclanthology.org/N18-1169.

Kenneth Li, Aspen K Hopkins, David Bau, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Emergent world representations: Exploring a sequence model trained on a synthetic task. *ArXiv preprint*, abs/2210.13382, 2022. URL https://arxiv.org/abs/2210.13382.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level Code Generation with AlphaCode. *ArXiv preprint*, abs/2105.12655, 2021b. URL https://arxiv.org/abs/2105.12655.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1823–1840, Online, 2020a. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.165. URL https://aclanthology.org/2020.findings-emnlp.165.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1823–1840, Online, 2020b. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.165. URL https://aclanthology.org/2020.findings-emnlp.165.

Chen Lin, Dmitriy Dligach, Timothy A Miller, Steven Bethard, and Guergana K Savova. Multilayered temporal modeling for the clinical domain. *Journal of the American Medical Informatics Association*, 23(2):387–395, 2016.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems. *ArXiv preprint*, abs/1705.04146, 2017. URL https://arxiv.org/abs/1705.04146.

Xiao Ling and Daniel S. Weld. Temporal information extraction. In Maria Fox and David Poole, editors, *Proc. of AAAI*. AAAI Press, 2010. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1805.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online, 2022a. Association for Computational Linguistics. doi: 10.18653/v1/2022.deelio-1.10. URL https://aclanthology.org/2022.deelio-1.10.

Jiacheng Liu, Skyler Hallinan, Ximing Lu, Pengfei He, Sean Welleck, Hannaneh Hajishirzi, and

Yejin Choi. Rainier: Reinforced knowledge introspector for commonsense question answering. In *Proc. of EMNLP*, pages 8938–8958, Abu Dhabi, United Arab Emirates, 2022b. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.611.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ArXiv*, 2021a.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ArXiv preprint*, abs/2107.13586, 2021b. URL https://arxiv.org/abs/2107.13586.

Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. FastBERT: a self-distilling BERT with adaptive inference time. In *Proc. of ACL*, pages 6035–6044, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.537. URL https://aclanthology.org/2020.acl-main.537.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *ArXiv preprint*, abs/1907.11692, 2019. URL https://arxiv.org/abs/1907.11692.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. *ArXiv preprint*, abs/2301.13688, 2023. URL https://arxiv.org/abs/2301.13688.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *ArXiv preprint*, abs/1711.05101, 2017. URL https://arxiv.org/abs/1711.05101.

Ximing Lu, Sean Welleck, Liwei Jiang, Jack Hessel, Lianhui Qin, Peter West, Prithviraj Ammanabrolu, and Yejin Choi. Quark: Controllable text generation with reinforced unlearning. *ArXiv preprint*, abs/2205.13636, 2022. URL https://arxiv.org/abs/2205.13636.

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proc. of EMNLP*, pages 1412–1421, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1166. URL https://aclanthology.org/D15-1166.

Shangwen Lv, Daya Guo, Jingjing Xu, Duyu Tang, Nan Duan, Ming Gong, Linjun Shou, Daxin Jiang, Guihong Cao, and Songlin Hu. Graph-based reasoning over heterogeneous external knowledge for commonsense question answering. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 8449–8456. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/6364.

Kaixin Ma, Filip Ilievski, Jonathan Francis, Eric Nyberg, and Alessandro Oltramari. Coalescing

global and local information for procedural text understanding. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 1534–1545, Gyeongju, Republic of Korea, 2022. International Committee on Computational Linguistics. URL https://aclanthology.org/2022.coling-1.132.

Aman Madaan and Yiming Yang. Neural language modeling for contextualized temporal graph generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 864–881, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.67. URL https://aclanthology.org/2021.naacl-main.67.

Aman Madaan and Amir Yazdanbakhsh. Text and patterns: For effective chain of thought, it takes two to tango. *ArXiv preprint*, abs/2209.07686, 2022a. URL https://arxiv.org/abs/2209.07686.

Aman Madaan and Amir Yazdanbakhsh. Text and patterns: For effective chain of thought, it takes two to tango. *ArXiv preprint*, abs/2209.07686, 2022b. URL https://arxiv.org/abs/2209.07686.

Aman Madaan, Dheeraj Rajagopal, Niket Tandon, Yiming Yang, and Eduard Hovy. Could you give me a hint ? generating inference graphs for defeasible reasoning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 5138–5147, Online, 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.456. URL https://aclanthology.org/2021.findings-acl.456.

Aman Madaan, Dheeraj Rajagopal, Niket Tandon, Yiming Yang, and Eduard Hovy. Could you give me a hint ? generating inference graphs for defeasible reasoning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 5138–5147, Online, 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.456. URL https://aclanthology.org/2021.findings-acl.456.

Aman Madaan, Niket Tandon, Dheeraj Rajagopal, Peter Clark, Yiming Yang, and Eduard Hovy. Think about it! improving defeasible reasoning by first modeling the question scenario. In *Proc. of EMNLP*, pages 6291–6310, Online and Punta Cana, Dominican Republic, 2021c. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.508. URL https://aclanthology.org/2021.emnlp-main.508.

Aman Madaan, Niket Tandon, Dheeraj Rajagopal, Peter Clark, Yiming Yang, and Eduard Hovy. Think about it! improving defeasible reasoning by first modeling the question scenario. In *Proc. of EMNLP*, pages 6291–6310, Online and Punta Cana, Dominican Republic, 2021d. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.508. URL https://aclanthology.org/2021.emnlp-main.508.

Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. Memory-assisted prompt editing to improve gpt-3 after deployment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2833–2861, 2022a.

Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. Memory-assisted prompt editing to improve GPT-3 after deployment. In *Proc. of EMNLP*, pages 2833–2861, Abu Dhabi, United Arab Emirates, 2022b. Association for Computational Linguistics. URL https:

//aclanthology.org/2022.emnlp-main.183.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In *Proc. of EMNLP*, pages 1384–1403, Abu Dhabi, United Arab Emirates, 2022c. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.90.

Aman Madaan, Pranjal Aggarwal, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. Automix: Automatically mixing language models. *ArXiv preprint*, abs/2310.12963, 2023a. URL https://arxiv.org/abs/2310.12963.

Aman Madaan, Katherine Hermann, and Amir Yazdanbakhsh. What makes chain-of-thought prompting effective? a counterfactual study. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1448–1535, 2023b.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits. *ArXiv preprint*, abs/2302.07867, 2023c. URL https://arxiv.org/abs/2302.07867.

Juha Makkonen, Helena Ahonen-Myka, and Marko Salmenkivi. Topic detection and tracking with spatio-temporal evidence. In *European Conference on Information Retrieval*, pages 251–265. Springer, 2003.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster Sorting Algorithms Discovered using Deep Reinforcement Learning. *Nature*, 2023.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. Generation-augmented retrieval for open-domain question answering. In *Proc. of ACL*, pages 4089–4100, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.316. URL https://aclanthology.org/2021.acl-long.316.

Gary Marcus. Deep learning: A critical appraisal. *ArXiv preprint*, abs/1801.00631, 2018. URL https://arxiv.org/abs/1801.00631.

Gary Marcus. The next decade in ai: four steps towards robust artificial intelligence. *ArXiv preprint*, abs/2002.06177, 2020. URL https://arxiv.org/abs/2002.06177.

Gary Marcus. Experiments testing gpt-3's ability at commonsense reasoning: results., 2021. URL https://cs.nyu.edu/~davise/papers/GPT3CompleteTests.html.

Andrew McCallum, Xuerui Wang, and Andrés Corrada-Emmanuel. Topic and role discovery in social networks with experiments on enron and academic email. *J. Artif. Int. Res.*, 30(1):249–272, 2007. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622637.1622644.

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

Shikib Mehri and Maxine Eskenazi. Unsupervised evaluation of interactive dialog with DialoGPT. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and*

*Dialogue*, pages 225–235, 1st virtual meeting, 2020. Association for Computational Linguistics. URL https://aclanthology.org/2020.sigdial-1.28.

Ardith J Meier. Defining politeness: Universality in appropriateness. *Language Sciences*, 17(4): 345–356, 1995.

Rui Meng, Sanqiang Zhao, Shuguang Han, Daqing He, Peter Brusilovsky, and Yu Chi. Deep keyphrase generation. In *Proc. of ACL*, pages 582–592, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1054. URL https://aclanthology.org/P17-1054.

Rui Meng, Xingdi Yuan, Tong Wang, Peter Brusilovsky, Adam Trischler, and Daqing He. Does order matter? an empirical study on generating multiple keyphrases as a sequence. *ArXiv preprint*, abs/1909.03590, 2019. URL https://arxiv.org/abs/1909.03590.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. In *Proc. of ICLR*. OpenReview.net, 2018. URL https://openreview.net/forum?id=SyyGPP0TZ.

Shen-yun Miao, Chao-Chun Liang, and Keh-Yih Su. A diverse corpus for evaluating and developing English math word problem solvers. In *Proc. of ACL*, pages 975–984, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.92. URL https://aclanthology.org/2020.acl-main.92.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, and Ashwin Kalyan. LILA: A unified benchmark for mathematical reasoning. In *Proc. of EMNLP*, pages 5807–5832, Abu Dhabi, United Arab Emirates, 2022a. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.392.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. Reframing instructional prompts to GPTk's language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 589–612, Dublin, Ireland, 2022b. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.50. URL https://aclanthology.org/2022.findings-acl.50.

Meredith Ringel Morris, Jascha Sohl-dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of agi: Operationalizing progress on the path to agi. *ArXiv preprint*, abs/2311.02462, 2023. URL https://arxiv.org/abs/2311.02462.

Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and evaluation framework for deeper understanding of commonsense stories. *ArXiv preprint*, abs/1604.01696, 2016. URL https://arxiv.org/abs/1604.01696.

Sreyasi Nag Chowdhury, Niket Tandon, and Gerhard Weikum. Know2Look: Commonsense knowledge for visual search. In *Proceedings of the 5th Workshop on Automated Knowledge Base Construction*, pages 57–62, San Diego, CA, 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-1311. URL https://aclanthology.org/W16-1311.

Allen Newell, Herbert Alexander Simon, et al. *Human problem solving*, volume 104. Prentice-hall Englewood Cliffs, NJ, 1972.

Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

Kim Anh Nguyen, Sabine Schulte im Walde, and Ngoc Thang Vu. Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction. In *Proc. of ACL*, pages 454–459, Berlin, Germany, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-2074. URL https://aclanthology.org/P16-2074.

Artur Niederfahrenhorst, Kourosh Hakhamaneshi, and Rehaan Ahmad. Fine-Tuning LLMs: LoRA or Full-Parameter? An In-Depth Analysis with Llama 2, 2023. Blog post.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint*, 2022a.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ArXiv preprint*, abs/2203.13474, 2022b. URL https://arxiv.org/abs/2203.13474.

Qiang Ning, Zhili Feng, and Dan Roth. A structured learning approach to temporal relation extraction. In *Proc. of EMNLP*, pages 1027–1037, Copenhagen, Denmark, 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1108. URL https://aclanthology.org/D17-1108.

Qiang Ning, Ben Zhou, Zhili Feng, Haoruo Peng, and Dan Roth. CogCompTime: A tool for understanding time in natural language. In *Proc. of EMNLP*, pages 72–77, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2013. URL https://aclanthology.org/D18-2013.

Qiang Ning, Hao Wu, Rujun Han, Nanyun Peng, Matt Gardner, and Dan Roth. TORQUE: A reading comprehension dataset of temporal ordering questions. In *Proc. of EMNLP*, pages 1158–1172, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.88. URL https://aclanthology.org/2020.emnlp-main.88.

Tong Niu and Mohit Bansal. Polite dialogue generation without parallel data. *Transactions of the Association for Computational Linguistics*, 6:373–389, 2018. doi: 10.1162/tacl_a_00027. URL https://aclanthology.org/Q18-1027.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. *ArXiv preprint*, abs/2102.13019, 2021. URL https://arxiv.org/abs/2102.13019.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your Work: Scratchpads for Intermediate Computation with Language Models. *ArXiv preprint*, abs/2112.00114, 2021a. URL https://arxiv.org/abs/2112.

00114.

Maxwell I. Nye, Michael Henry Tessler, Joshua B. Tenenbaum, and Brenden M. Lake. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 25192–25204, 2021b. URL https://proceedings.neurips.cc/paper/2021/hash/d3e2e8f631bd9336ed25b8162aef8782-Abstract.html.

Augustus Odena and Charles Sutton. Learning to Represent Programs with Property Signatures. *ArXiv preprint*, abs/2002.09030, 2020. URL https://arxiv.org/abs/2002.09030.

OpenAI. Model index for researchers. https://platform.openai.com/docs/model-index-for-researchers. Accessed: May 14, 2023.

OpenAI. Openai completion engine (davinci) api, 2021. URL https://beta.openai.com/docs/guides/completion.

OpenAI. Model index for researchers, 2022. URL https://beta.openai.com/docs/model-index-for-researchers. Blogpost.

OpenAI. Gpt-4 technical report, 2023.

Pedro A Ortega, Markus Kunesch, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Joel Veness, Jonas Buchli, Jonas Degrave, Bilal Piot, Julien Perolat, et al. Shaking the foundations: delusions in sequence models for interaction and control. *ArXiv preprint*, abs/2110.10819, 2021. URL https://arxiv.org/abs/2110.10819.

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 E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv preprint*, abs/2203.02155, 2022a. URL https://arxiv.org/abs/2203.02155.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022b.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proc. of ACL*, pages 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Workshop Autodiff Submission*, 2017.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP Models Really Able to Solve Simple Math Word Problems? *ArXiv preprint*, abs/2103.07191, 2021. URL https://arxiv.

org/abs/2103.07191.

Judea Pearl et al. Models, reasoning and inference. *Cambridge, UK: CambridgeUniversityPress*, 2000.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. Check Your Facts and Try Again: Improving Large Language Models with External Knowledge and Automated Feedback. *ArXiv preprint*, abs/2302.12813, 2023. URL https://arxiv.org/abs/2302.12813.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proc. of EMNLP*, pages 1532–1543, Doha, Qatar, 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://aclanthology.org/D14-1162.

David Peter. hyperfine, 2023. URL https://github.com/sharkdp/hyperfine.

Kelly Peterson, Matt Hohensee, and Fei Xia. Email formality in the workplace: A case study on the Enron corpus. In *Proceedings of the Workshop on Language in Social Media (LSM 2011)*, pages 86–95, Portland, Oregon, 2011. Association for Computational Linguistics. URL https://aclanthology.org/W11-0711.

Xinyu Pi, Qian Liu, Bei Chen, Morteza Ziyadi, Zeqi Lin, Qiang Fu, Yan Gao, Jian-Guang Lou, and Weizhu Chen. Reasoning like program executors. In *Proc. of EMNLP*, pages 761–779, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.48.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *Proc. of ICLR*. OpenReview.net, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

J. Pollock. Defeasible reasoning. *Cogn. Sci.*, 11:481–518, 1987.

J. Pollock. A recursive semantics for defeasible reasoning. In *Argumentation in Artificial Intelligence*, 2009.

Michael I. Posner and Charles R. Snyder. Attention and cognitive control., 1975.

Vinodkumar Prabhakaran, Emily E. Reid, and Owen Rambow. Gender and power: How gender and gender environment affect manifestations of power. In *Proc. of EMNLP*, pages 1965–1976, Doha, Qatar, 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1211. URL https://aclanthology.org/D14-1211.

Shrimai Prabhumoye, Yulia Tsvetkov, Ruslan Salakhutdinov, and Alan W Black. Style transfer through back-translation. In *Proc. of ACL*, pages 866–876, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1080. URL https://aclanthology.org/P18-1080.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8494–8502. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.

00886. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Puig_VirtualHome_Simulating_Household_CVPR_2018_paper.html.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *ArXiv preprint*, abs/2105.12655, 2021a. URL https://arxiv.org/abs/2105.12655.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *ArXiv preprint*, abs/2105.12655, 2021b. URL https://arxiv.org/abs/2105.12655.

Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. Limitations of language models in arithmetic and symbolic induction. *ArXiv preprint*, abs/2208.05051, 2022. URL https://arxiv.org/abs/2208.05051.

Kechen Qin, Cheng Li, Virgil Pavlu, and Javed Aslam. Adapting RNN sequence prediction model to multi-label set prediction. In *Proc. of NAACL-HLT*, pages 3181–3190, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1321. URL https://aclanthology.org/N19-1321.

Alec Radford. Improving language understanding by generative pre-training, 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020a. URL http://jmlr.org/papers/v21/20-074.html.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020b. URL http://jmlr.org/papers/v21/20-074.html.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020c. URL http://jmlr.org/papers/v21/20-074.html.

Dheeraj Rajagopal, Aman Madaan, Niket Tandon, Yiming Yang, Shrimai Prabhumoye, Abhilasha Ravichander, Peter Clark, and Eduard H Hovy. CURIE: An iterative querying approach for reasoning about situations. In *Proceedings of the First Workshop on Commonsense Representation and Reasoning (CSRR 2022)*, pages 49–63, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.csrr-1.7. URL https://aclanthology.org/2022.csrr-1.7.

Sudha Rao and Joel Tetreault. Dear sir or madam, may I introduce the GYAFC dataset: Corpus, benchmarks and metrics for formality style transfer. In *Proc. of NAACL-HLT*, pages 129–140, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1012. URL https://aclanthology.org/N18-1012.

Alexander J. Ratner, Henry R. Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. Learning to compose domain-specific transformations for data augmentation. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3236–3246, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/f26dab9bf6a137c3b6782e562794c2f2-Abstract.html.

Sravana Reddy and Kevin Knight. Obfuscating gender in social media writing. In *Proceedings of the First Workshop on NLP and Computational Social Science*, pages 17–26, Austin, Texas, 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-5603. URL https://aclanthology.org/W16-5603.

Machel Reid and Graham Neubig. Learning to model editing processes. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3822–3832, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.280.

Emily Reif, Daphne Ippolito, Ann Yuan, Andy Coenen, Chris Callison-Burch, and Jason Wei. A Recipe for Arbitrary Text Style Transfer with Large Language Models. *ArXiv preprint*, abs/2109.03910, 2021. URL https://arxiv.org/abs/2109.03910.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proc. of EMNLP*, pages 3982–3992, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410.

S Hamid Rezatofighi, Roman Kaskman, Farbod T Motlagh, Qinfeng Shi, Daniel Cremers, Laura Leal-Taixé, and Ian Reid. Deep perm-set net: Learn to predict sets with unknown permutation and cardinality using deep neural networks. *ArXiv preprint*, abs/1805.00613, 2018. URL https://arxiv.org/abs/1805.00613.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *ArXiv preprint*, abs/2401.08500, 2024. URL https://arxiv.org/abs/2401.08500.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *ArXiv preprint*, abs/2308.12950, 2023. URL https://arxiv.org/abs/2308.12950.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context

learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, Seattle, United States, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.191. URL https://aclanthology.org/2022.naacl-main.191.

Rachel Rudinger, Vered Shwartz, Jena D. Hwang, Chandra Bhagavatula, Maxwell Forbes, Ronan Le Bras, Noah A. Smith, and Yejin Choi. Thinking like a skeptic: Defeasible inference in natural language. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4661–4675, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.418. URL https://aclanthology.org/2020.findings-emnlp.418.

Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

Swarnadeep Saha, Prateek Yadav, Lisa Bauer, and Mohit Bansal. ExplaGraphs: An explanation graph generation task for structured commonsense reasoning. In *Proc. of EMNLP*, pages 7716–7740, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.609. URL https://aclanthology.org/2021.emnlp-main.609.

Keisuke Sakaguchi, Chandra Bhagavatula, Ronan Le Bras, Niket Tandon, Peter Clark, and Yejin Choi. proscript: Partially ordered scripts generation via pre-trained language models. *arxiv*, 2021a.

Keisuke Sakaguchi, Chandra Bhagavatula, Ronan Le Bras, Niket Tandon, Peter Clark, and Yejin Choi. proScript: Partially ordered scripts generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2138–2149, Punta Cana, Dominican Republic, 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.184. URL https://aclanthology.org/2021.findings-emnlp.184.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Stella Biderman, Leo Gao, Tali Bers, Thomas Wolf, and Alexander M. Rush. Multitask Prompted Training Enables Zero-Shot Task Generalization, 2021. URL https://arxiv.org/abs/2110.08207.

Maarten Sap, Ronan Le Bras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A. Smith, and Yejin Choi. ATOMIC: an atlas of machine commonsense for if-then reasoning. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 3027–3035. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33013027. URL https://doi.org/10.1609/aaai.v33i01.33013027.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan

Leike. Self-critiquing models for assisting human evaluators. *ArXiv preprint*, abs/2206.05802, 2022a. URL https://arxiv.org/abs/2206.05802.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators. *ArXiv preprint*, abs/2206.05802, 2022b. URL https://arxiv.org/abs/2206.05802.

Jérémy Scheurer, Jon Ander Campos, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. Training language models with natural language feedback. *ArXiv preprint*, abs/2204.14146, 2022. URL https://arxiv.org/abs/2204.14146.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. Peer: A collaborative language model. *ArXiv preprint*, abs/2208.11663, 2022a. URL https://arxiv.org/abs/2208.11663.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. Peer: A collaborative language model. *ArXiv preprint*, abs/2208.11663, 2022b. URL https://arxiv.org/abs/2208.11663.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015.

Tal Schuster, Adam Fisch, Tommi Jaakkola, and Regina Barzilay. Consistent accelerated inference via confident adaptive transformers. In *Proc. of EMNLP*, pages 4962–4979, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.406. URL https://aclanthology.org/2021.emnlp-main.406.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. of ACL*, pages 1715–1725, Berlin, Germany, 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Murray Shanahan. *The technological singularity*. MIT press, 2015.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv preprint*, abs/2402.03300, 2024. URL https://arxiv.org/abs/2402.03300.

Shikhar Sharma, Layla El Asri, Hannes Schulz, and Jeremie Zumer. Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation. *ArXiv preprint*, abs/1706.09799, 2017. URL https://arxiv.org/abs/1706.09799.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proc. of ICLR*. OpenReview.net, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Tianxiao Shen, Tao Lei, Regina Barzilay, and Tommi S. Jaakkola. Style transfer from non-parallel text by cross-alignment. In Isabelle Guyon, Ulrike von Luxburg, Samy Ben-

gio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6830–6841, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/2d2c8394e31101a261abf1784302bf75-Abstract.html.

Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, 4(1): 120–128, 2004.

M. Shi, Yufei Tang, Xingquan Zhu, and J. Liu. Feature-attention graph convolutional networks for noise resilient learning. *ArXiv preprint*, abs/1912.11755, 2019. URL https://arxiv.org/abs/1912.11755.

Richard M. Shiffrin and Walter Schneider. Controlled and automatic human information processing: Ii. perceptual learning, automatic attending and a general theory. *Psychological Review*, 84:127–190, 1977.

Richard Shin and Benjamin Van Durme. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5417–5425, Seattle, United States, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.396. URL https://aclanthology.org/2022.naacl-main.396.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. Constrained language models yield few-shot semantic parsers. In *Proc. of EMNLP*, pages 7699–7715, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.608. URL https://aclanthology.org/2021.emnlp-main.608.

Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection, 2023.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code. In *ICML*, 2023.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *ArXiv preprint*, abs/2302.07867, 2023. URL https://arxiv.org/abs/2302.07867.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587): 484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv preprint*, abs/1712.01815, 2017. URL https://arxiv.org/abs/1712.01815.

Herbert A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962. ISSN 0003049X. URL http://www.jstor.org/stable/985254.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Keith E Stanovich. Individual differences in reasoning: Implications for the rationality debate? *BEHAVIORAL AND BRAIN SCIENCES*, page 82, 2000.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F. Christiano. Learning to summarize with human feedback. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1f89885d556929e98d3ef9b86448f951-Abstract.html.

Akhilesh Sudhakar, Bhargav Upadhyay, and Arjun Maheswaran. "transforming" delete, retrieve, generate approach for controlled text style transfer. In *Proc. of EMNLP*, pages 3269–3279, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1322. URL https://aclanthology.org/D19-1322.

Douglas Summers-Stay, Claire Bonial, and Clare Voss. What can a generative language model answer about a passage? In *Proceedings of the 3rd Workshop on Machine Reading for Question Answering*, pages 73–81, Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.mrqa-1.7. URL https://aclanthology.org/2021.mrqa-1.7.

Zhiqing Sun, Yikang Shen, Qinhong Zhou, Hongxin Zhang, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. Principle-driven self-alignment of language models from scratch with minimal human supervision. *ArXiv preprint*, abs/2305.03047, 2023. URL https://arxiv.org/abs/2305.03047.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014. URL https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

R. S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. Accessed: March 7, 2024.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *ArXiv preprint*, abs/2210.09261, 2022a. URL https://arxiv.org/abs/2210.09261.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won

Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *ArXiv preprint*, abs/2210.09261, 2022b. URL https://arxiv.org/abs/2210.09261.

Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

Alon Talmor, Oyvind Tafjord, Peter Clark, Yoav Goldberg, and Jonathan Berant. Leap-of-thought: Teaching pre-trained models to systematically reason over implicit knowledge. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/e992111e4ab9985366e806733383bd8c-Abstract.html.

Niket Tandon, Bhavana Dalvi, Keisuke Sakaguchi, Peter Clark, and Antoine Bosselut. WIQA: A dataset for "what if..." reasoning over procedural text. In *Proc. of EMNLP*, pages 6076–6085, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1629. URL https://aclanthology.org/D19-1629.

Niket Tandon, Aman Madaan, Peter Clark, Keisuke Sakaguchi, and Yiming Yang. Interscript: A dataset for interactive learning of scripts through error feedback. *ArXiv preprint*, abs/2112.07867, 2021. URL https://arxiv.org/abs/2112.07867.

Niket Tandon, Aman Madaan, Peter Clark, and Yiming Yang. Learning to repair: Repairing model output errors after deployment using a dynamic memory of feedback. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 339–352, Seattle, United States, 2022a. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.26. URL https://aclanthology.org/2022.findings-naacl.26.

Niket Tandon, Aman Madaan, Peter Clark, and Yiming Yang. Learning to repair: Repairing model output errors after deployment using a dynamic memory of feedback. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 339–352, Seattle, United States, 2022b. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.26. URL https://aclanthology.org/2022.findings-naacl.26.

Till Tantau. Graph drawing in TikZ. In *Proceedings of the 20th International Conference on Graph Drawing*, GD'12, pages 517–528, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36762-5. doi: 10.1007/978-3-642-36763-2_46.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *ArXiv preprint*, abs/2312.11805, 2023. URL https://arxiv.org/abs/2312.11805.

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

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Naushad UzZaman and James F Allen. *Interpreting the temporal aspects of language*. Citeseer, 2012.

Naushad UzZaman, Hector Llorens, Leon Derczynski, James Allen, Marc Verhagen, and James Pustejovsky. SemEval-2013 task 1: TempEval-3: Evaluating time expressions, events, and temporal relations. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, pages 1–9, Atlanta, Georgia, USA, 2013. Association for Computational Linguistics. URL https://aclanthology.org/S13-2001.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. In Yoshua Bengio and Yann LeCun, editors, *Proc. of ICLR*, 2016. URL http://arxiv.org/abs/1511.06391.

Rob Voigt, David Jurgens, Vinodkumar Prabhakaran, Dan Jurafsky, and Yulia Tsvetkov. RtGender: A corpus for studying differential responses to gender. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1445.

M Mitchell Waldrop. *The dream machine*. Stripe Press, 2018.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 3261–3275, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/4496bf24afe7fab6f046bf4923da8de6-Abstract.html.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Rationale-Augmented Ensembles in Language Models. *ArXiv preprint*, abs/2207.00747, 2022a. URL https://arxiv.org/abs/2207.00747.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *ArXiv preprint*, abs/2203.11171, 2022b. URL https://arxiv.org/abs/2203.11171.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware uni-

fied pre-trained encoder-decoder models for code understanding and generation. In *Proc. of EMNLP*, pages 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Lilian D. A. Wanzare, Alessandra Zarcone, Stefan Thater, and Manfred Pinkal. A crowdsourced database of event sequence descriptions for the acquisition of high-quality script knowledge. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 3494–3501, Portorož, Slovenia, 2016. European Language Resources Association (ELRA). URL https://aclanthology.org/L16-1556.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned Language Models are Zero-shot Learners. *ArXiv preprint*, abs/2109.01652, 2021. URL https://arxiv.org/abs/2109.01652.

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. Emergent Abilities of Large Language Models. *ArXiv preprint*, abs/2206.07682, 2022a. URL https://arxiv.org/abs/2206.07682.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv preprint*, abs/2201.11903, 2022b. URL https://arxiv.org/abs/2201.11903.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv preprint*, abs/2201.11903, 2022c. URL https://arxiv.org/abs/2201.11903.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv preprint*, abs/2201.11903, 2022d. URL https://arxiv.org/abs/2201.11903.

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *Proc. of ICLR*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SJeYe0NtvH.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. *ArXiv preprint*, abs/2211.00053, 2022. URL https://arxiv.org/abs/2211.00053.

Manfred Wettler and Reinhard Rapp. Computation of word associations based on co-occurrences of words in large corpora. In *Very Large Corpora: Academic and Industrial Perspectives*, 1993. URL https://aclanthology.org/W93-0310.

Thomas Wolf, L Debut, V Sanh, J Chaumond, C Delangue, A Moi, P Cistac, T Rault, R Louf, M Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv preprint*, abs/1910.03771, 2019. URL https://arxiv.org/abs/1910.03771.

Xing Wu, Tao Zhang, Liangjun Zang, Jizhong Han, and Songlin Hu. Mask and infill: Applying masked language model for sentiment transfer. In Sarit Kraus, editor, *Proceedings of the*

*Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 5271–5277. ijcai.org, 2019. doi: 10.24963/ijcai.2019/732. URL https://doi.org/10.24963/ijcai.2019/732.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022a.

Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *Proc. of ICLR*. OpenReview.net, 2022b. URL https://openreview.net/forum?id=TrjbxzRcnf-.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *ArXiv preprint*, abs/2202.13169, 2022a. URL https://arxiv.org/abs/2202.13169.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A Systematic Evaluation of Large Language Models of Code. In *MAPS*, 2022b.

Ruochen Xu, Tao Ge, and Furu Wei. Formality style transfer with hybrid textual annotations. *ArXiv preprint*, abs/1903.06353, 2019. URL https://arxiv.org/abs/1903.06353.

Wei Xu, Alan Ritter, Bill Dolan, Ralph Grishman, and Colin Cherry. Paraphrasing for style. In *Proceedings of COLING 2012*, pages 2899–2914, Mumbai, India, 2012. The COLING 2012 Organizing Committee. URL https://aclanthology.org/C12-1177.

Kevin Yang, Yuandong Tian, Nanyun Peng, and Dan Klein. Re3: Generating longer stories with recursive reprompting and revision. In *Proc. of EMNLP*, pages 4393–4479, Abu Dhabi, United Arab Emirates, 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.296.

Pengcheng Yang, Xu Sun, Wei Li, Shuming Ma, Wei Wu, and Houfeng Wang. SGM: Sequence generation model for multi-label classification. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3915–3926, Santa Fe, New Mexico, USA, 2018a. Association for Computational Linguistics. URL https://aclanthology.org/C18-1330.

Yiben Yang, Chaitanya Malaviya, Jared Fernandez, Swabha Swayamdipta, Ronan Le Bras, Ji-Ping Wang, Chandra Bhagavatula, Yejin Choi, and Doug Downey. G-daug: Generative data augmentation for commonsense reasoning. In *Proc. of EMNLP*, pages 1008–1025, 2020.

Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 42–49, 1999.

Zichao Yang, Zhiting Hu, Chris Dyer, Eric P. Xing, and Taylor Berg-Kirkpatrick. Unsupervised text style transfer using language models as discriminators. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7298–7309, 2018b. URL https://proceedings.neurips.cc/paper/

2018/hash/398475c83b47075e8897a083e97eb9f0-Abstract.html.

Thomas Wolf Yangfeng Ji, Antoine Bosselut and Asli Celikyilmaz. The amazing world of generation. *EMNLP tutorials*, 2020. URL https://nlg-world.github.io/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv preprint*, abs/2210.03629, 2022. URL https://arxiv.org/abs/2210.03629.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *Proc. of ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 10799–10808. PMLR, 2020. URL http://proceedings.mlr.press/v119/yasunaga20a.html.

Jiacheng Ye, Tao Gui, Yichao Luo, Yige Xu, and Qi Zhang. One2Set: Generating diverse keyphrases as a set. In *Proc. of ACL*, pages 4598–4608, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.354. URL https://aclanthology.org/2021.acl-long.354.

Jen-Yuan Yeh and Aaron Harnly. Email thread reassembly using similarity matching. In *Conference on Email and Anti-Spam*. Conference on Email and Anti-Spam, 2006.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In Jennifer G. Dy and Andreas Krause, editors, *Proc. of ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5694–5703. PMLR, 2018. URL http://proceedings.mlr.press/v80/you18a.html.

Xingdi Yuan, Tong Wang, Rui Meng, Khushboo Thaker, Peter Brusilovsky, Daqing He, and Adam Trischler. One size does not fit all: Generating and evaluating variable number of keyphrases. In *Proc. of ACL*, pages 7961–7975, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.710. URL https://aclanthology.org/2020.acl-main.710.

Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Proc. of ACL*, pages 2204–2213, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1205. URL https://aclanthology.org/P18-1205.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. AMR parsing as sequence-to-graph transduction. In *Proc. of ACL*, pages 80–94, Florence, Italy, 2019a. Association for Computational Linguistics. doi: 10.18653/v1/P19-1009. URL https://aclanthology.org/P19-1009.

Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E Gonzalez. The Wisdom of Hindsight Makes Language Models Better Instruction Followers. *ArXiv preprint*, abs/2302.05206, 2023. URL https://arxiv.org/abs/2302.05206.

Tianjun Zhang, Aman Madaan, Luyu Gao, Steven Zheng, Swaroop Mishra, Yiming Yang, Niket Tandon, and Uri Alon. In-context principle learning from mistakes. *ArXiv preprint*, abs/2402.05403, 2024. URL https://arxiv.org/abs/2402.05403.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore:

Evaluating text generation with BERT. In *Proc. of ICLR*. OpenReview.net, 2020a. URL https://openreview.net/forum?id=SkeHuCVFDr.

Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html.

Yan Zhang, Jonathon S. Hare, and Adam Prügel-Bennett. Deep set prediction networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 3207–3217, 2019b. URL https://proceedings.neurips.cc/paper/2019/hash/6e79ed05baec2754e25b4eac73a332d2-Abstract.html.

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. DIALOGPT : Large-scale generative pre-training for conversational response generation. In *Proc. of ACL*, pages 270–278, Online, 2020b. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-demos.30. URL https://aclanthology.org/2020.acl-demos.30.

Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. ERNIE: Enhanced language representation with informative entities. In *Proc. of ACL*, pages 1441–1451, Florence, Italy, 2019c. Association for Computational Linguistics. doi: 10.18653/v1/P19-1139. URL https://aclanthology.org/P19-1139.

Wenting Zhao, Mor Geva, Bill Yuchen Lin, Michihiro Yasunaga, Aman Madaan, and Tao Yu. Complex reasoning in natural language. In *Proc. of ACL*, pages 11–20, 2023.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving Few-shot Performance of Language Models. In *ICML*, 2021.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. LIMA: Less Is More for Alignment. *ArXiv preprint*, abs/2305.11206, 2023a. URL https://arxiv.org/abs/2305.11206.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. *ArXiv preprint*, abs/2205.10625, 2022. URL https://arxiv.org/abs/2205.10625.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. *ArXiv preprint*, abs/2302.05527, 2023b. URL https://arxiv.org/abs/2302.05527.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *ArXiv preprint*, abs/2307.13854, 2023c. URL https://arxiv.org/

abs/2307.13854.

Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian J. McAuley, Ke Xu, and Furu Wei. BERT loses patience: Fast and robust inference with early exit. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/d4dd111a4fd973394238aca5c05bebe3-Abstract.html.

# Appendices

# A Chapter 2: Neural Language Modeling for Contextualized Temporal Graph Generation

## A.1 Learning Event Communities Using Community Detection

In this section, we provide the details on the community detection algorithm used by our method. We define the temporal event communities to be a division of the temporal graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ into sub-graphs $\mathbf{G_1}(\mathbf{V_1}, \mathbf{E_1}), \mathbf{G_2}(\mathbf{V_2}, \mathbf{E_2}), ..., \mathbf{G_k}(\mathbf{V_k}, \mathbf{E_k})$ such that the events in a community (sub-graph) $\mathbf{G_i}$ are more co-referential to each other as opposed to the other events in the temporal graph. We use the undirected link between two events $e_j, e_i$ as a proxy for them being co-referential, and learn temporal event communities utilizing the concept of modularity, first introduced by Newman and Girvan [2004].

Formally, let $\mathbf{A}$ be the undirected adjacency matrix for a temporal graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ such that $\mathbf{A}(e_i, e_j) = 1$ if $e_i$ and $e_j$ are connected by a temporal relation, and 0 otherwise. Further, let $\delta(e_i, e_j) = 1$ if events $e_i, e_j$ belong to the same temporal community, and 0 otherwise. For a given $\delta$, we denote the fraction of the edges that exist between events that belong to the same communities by $f_{same} = \frac{\sum_{e_i,e_j \in \mathbf{E}} \mathbf{A}(e_i,e_j)\delta(e_i,e_j)}{2|\mathbf{E}|}$. Where the $2|\mathbf{E}|$ in the denominator is due to the fact that $\mathbf{A}$ treats $\mathbf{G}$ as an undirected graph. Let the popularity $p$ of an event $e_i$ be the number of events that are linked to it i.e. $p(e_i) = \sum_{e_j \in \mathbf{E}} \mathbf{A}(e_i, e_j)$. The probability of randomly picking an event $e_i$ when sampled by popularity is $\frac{p(e_i)}{\sum_{e_j \in \mathbf{E}} p(e_i)} = \frac{p(e_i)}{2|\mathbf{E}|}$. Thus, if edges are created randomly by sampling nodes by popularity $p$ of the nodes, the fraction of edges within the communities, $f_{random}$, is given by

$$f_{random} = \frac{\sum_{e_i,e_j \in \mathbf{E}} p(e_i)p(e_j)\delta(e_i, e_j)}{2|\mathbf{E}| * 2|\mathbf{E}|}$$

Finally, defining modularity, $Q$, to be $f_{same} - f_{random}$:

$$Q = \frac{1}{2|\mathbf{E}|} * \sum_{e_i,e_j \in \mathbf{E}} \mathbf{A}(e_i, e_j) - \frac{p(e_i)p(e_j)\delta(e_i, e_j)}{2|\mathbf{E}|}$$

We want to learn community assignments $\delta$ that maximize $Q$. A high $Q$ would promote $f_{same} > f_{random}$ and thereby encourage highly inter-connected event communities. Calculating such $\delta$ directly is not tractable, since the complexity of such an operation would be at least exponential in the number of events Newman [2004]. We use the fast implementation provided by Clauset et al. [2004] for calculating event communities iteratively. The algorithm converges at $Q$ 0.3. We use a similar approximation at test time: given a document $D$, we first break it down into sub-documents using CAEVO and then feed each sub-document to our method.

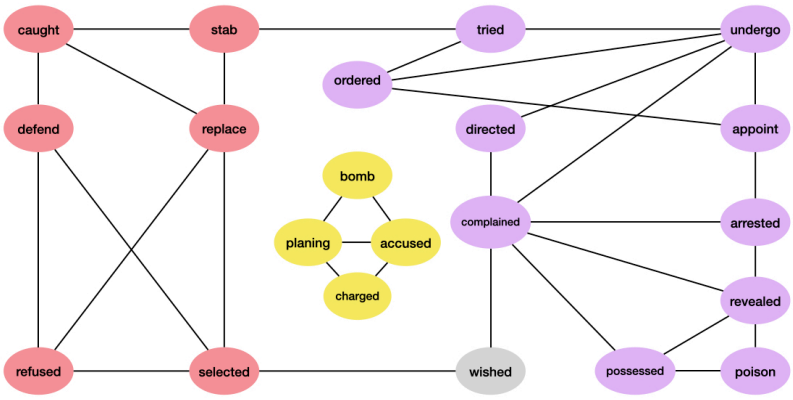



Figure 7: Event temporal graph and the extracted communities for a sample document. Each community is shown in different color. The singleton nodes (**gray**) are dropped. The nodes are only annotated with the verbs for brevity. The edge labels and directions are not used for community detection.

## A.2 Using a smaller block size

We found that the performance drops when using a block size of 300 and batch size of 2. Table 4 presents the results.

| BLEU | MTR | RG | DOT% |
|---|---|---|---|
| **25.01** | **27.95** | **60.99** | **91.71** |

| $v_P$ | $v_R$ | $v_{F_1}$ | $e_P$ | $e_R$ | $e_{F_1}$ |
|---|---|---|---|---|---|
| 70.31 | 64.75 | 65.68 | 29.43 | 24.83 | 24.27 |

Table 4: Results for TG-Gen using a block size of 300 and a block size of 2.

## A.3 Masked Language Modeling Using Transformers

In this section, we expand on the design of the transformer blocks. For ease of reference, we re-iterate our training methodology. We train a (separate) conditional language model to solve both the tasks. Specifically, given a training corpus of the form $\{(\S x_i, \S y_i)\}$, we aim to estimate the distribution $p_\theta(\S y_i \mid \S x_i)$. Given a training example $(\S x_i, \S y_i)$ we set $\S u_i = \S x_i \| \S y_i$[4]. $p_\theta(\S u_i)$ can then be factorized as a sequence of auto-regressive conditional probabilities using the chain rule: $p_\theta(\S u_i) = \prod_{k=1}^{n} p(u_{i,k} | \S u_{i,<k})$, where $u_{i,k}$ denotes the $k^{th}$ token of the $i^{th}$ sequence, and $\S u_{i,<k}$ denotes the sequence of tokens $\{u_1, u_2, ..., u_{k-1}\}$. Language models are typically trained by minimizing a cross-entropy loss $-\log p_\theta(\S u_i)$ over each sequence $\S u_i$ in $\mathbf{X}$. However, the

[4] $\|$ denotes concatenation

cross-entropy loss captures the joint distribution $p_\theta(\S x_i, \S y_i)$, and is not aligned with our goal of learning conditional distribution $p_\theta(\S y_i|\S x_i)$. To circumvent this, we train our model by masking the loss terms corresponding to the input $\S x_i$, similar to Bosselut et al. [2019]. Let $\S m_i$ be a mask vector for each sequence $\S u_i$, set to 0 for positions corresponding to $\S x_i$, and 1 otherwise i.e. $m_{i,j} = 1$ if $j > |\S x_i|$, else 0. We combine the mask vector with our factorization of $p_\theta(\S u_i)$ to formulate a *masked* language modeling loss, which is minimized over the training corpus $\mathbf{X}$ to estimate the optimal $\theta$:

$$\mathcal{L}_{masked}(\mathbf{X}) = -\sum_{i=1}^{|\mathbf{X}|} \sum_{j=1}^{|x_i|+|y_i|} m_{i,j} * \log(p_\theta(u_{i,j}|\S u_{i,<j}))$$

Note that the formulation of masked loss is opaque to the underlying architecture, and can be implemented with a simple change to the loss function. Intuitively, the model is optimized for only the output sequence $y_i$.

### Adapting GPT-2 for Masked Language Modeling

In practice, we use GPT-2 Radford et al. [2019] based on transformer architecture Vaswani et al. [2017] for our implementation. An input sequence $\S u_i$ of length $n$ is first embedded to a continuous representation denoted by $\S {u_i}^{(0)} \in \mathbb{R}^{nd}$. $\S {u_i}^{(0)}$ is then passed through a series of $L$ *transformer blocks* to obtain the output sequence $\S {u_i}^{(L)} \in \mathbb{R}^{nh}$. Each transformer block Vaswani et al. [2017] consists of two operations: an auto-regressive version of the multiheaded self-attention Vaswani et al. [2017] operation (*AutoRegMultiHead*) followed by a feed-forward operation (*FFN*). Each of these operations is surrounded by a residual connection He et al. [2016] and followed by a layer normalization Ba et al. [2016] operation. Denoting by $\S u^{(l-1)}$ the input to the $l^{th}$ transformer block , the operations are in a transformer block are defined as follows:

$$\begin{aligned}
\tilde{\S u}^{l}_{attn} &= \textit{AutoRegMultiHead}(\S u^{(l-1)}) \\
\S u^{(l)}_{att} &= \textit{LayerNorm}(\tilde{\S u}^{(l)}_{att} + \S u^{(l-1)}) \\
\tilde{\S u}^{(l)}_{ffn} &= \textit{FFN}(\S u^{(l)}_{att}) \\
\S u^{(l)} &= \textit{LayerNorm}(\tilde{\S u}^{(l)}_{ffn} + \S u^{(l)}_{att})
\end{aligned}$$

Where *AutoRegMultiHead* is an auto-regressive version of the multiheaded self-attention Vaswani et al. [2017] that restricts the attention to the sequence seen so far (in accordance with the chain rule), and *FFN* is a feed-forward network (MLP). After obtaining $\S {u_i}^{(L)}$, we set $p_\phi(\S u_i) = \text{softmax}(\S {u_i}^{(L)} * \mathbf{W}_e)$, where $\mathbf{W}_e \in \mathbb{R}^{h|V|}$ ($|V|$ is the size of the vocabulary). Finally, we calculate the masked loss as $\mathcal{L}(\S u_i) = \S {m_i}^T \odot \log(p_\phi(\S u_i))$, and the optimal $\phi$ is obtained by minimizing $\mathcal{L}_{masked}(\mathbf{X}) = -\sum_{i=1}^{|\mathbf{X}|} \mathcal{L}(\S u_i)$.

## A.4 Dataset Statistics

Tables 5, 6, and 7 list various statistics calculated from the source data.

| Descriptor | #Articles |
|---|---|
| terrorism | 40909 |
| murders and attempted murders | 25169 |
| united states international relations | 17761 |
| united states armament and defense | 16785 |
| airlines and airplanes | 16103 |
| world trade center (nyc) | 15145 |
| demonstrations and riots | 14477 |
| hijacking | 14472 |
| politics and government | 6270 |
| bombs and explosives | 5607 |

Table 5: Top Descriptors for the filtered Dataset. Note that each article is typically assigned more than one descriptor.

| Event verb | Raw frequency | % Frequency |
|---|---|---|
| said | 647685 | 9.60 |
| say | 57667 | 0.86 |
| had | 47320 | 0.70 |
| killed | 43369 | 0.64 |
| told | 42983 | 0.64 |
| found | 41733 | 0.62 |
| made | 40544 | 0.60 |
| war | 35257 | 0.52 |
| get | 30726 | 0.46 |
| make | 29407 | 0.44 |

Table 6: Most frequent events extracted by CAEVO.

### A.5 Examples

Figures 8-13 show randomly picked examples from the test corpus. Each figure shows the text, the corresponding true graph, and the graph predicted by GPT-2.

## B Chapter 3: Conditional Set Generation with SEQ2SEQ models

### B.1 Proofs

Let $\mathbb{Y}$ be the output space, $\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k \in \mathbb{Y}$, and $\boldsymbol{y}_k \in \mathbb{Y} - \mathrm{y}_i - \mathrm{y}_j$ be a subset of the symbols excluding $\mathrm{y}_i, \mathrm{y}_j$. We assume that all the distributions are non-negative (i.e., $p(\boldsymbol{y}) > 0, \forall \boldsymbol{y} \in \mathbb{Y}$)

**Lemma B.1.** $\mathrm{y}_i \not\perp\!\!\!\perp \mathrm{y}_j \implies \mathrm{y}_i \not\perp\!\!\!\perp (\mathrm{y}_j \mathrm{y}_k)$

| Relation | Raw Frequency | % Frequency |
|---|---|---|
| BEFORE | 2436201 | 54.51 |
| AFTER | 1772071 | 39.65 |
| IS INCLUDED | 131052 | 2.93 |
| SIMULTANEOUS | 112509 | 2.52 |
| INCLUDES | 17465 | 0.39 |

Table 7: Relation Frequence in our Corpus

| Relation | Frequency |
|---|---|
| BEFORE | 98715 |
| AFTER | 68582 |
| IS INCLUDED | 6179 |
| SIMULTANEOUS | 6209 |
| INCLUDES | 285 |

Table 8: Edges in Generated Graphs: Top

**Proof** Let $\mathrm{y}_i \perp\!\!\!\perp (\mathrm{y}_j \mathrm{y}_k)$ by contradiction. Then:

$$p(\mathrm{y}_i, \mathrm{y}_j \mathrm{y}_k) = p(\mathrm{y}_i) p(\mathrm{y}_j \mathrm{y}_k) \tag{5}$$

Also,

$$\begin{aligned} p(\mathrm{y}_i, \mathrm{y}_j) &= \sum_{\mathrm{y}_k \in \boldsymbol{Z}} p(\mathrm{y}_i, \mathrm{y}_j \mathrm{y}_k) \\ &= \sum_{\mathrm{y}_k \in \boldsymbol{Z}} p(\mathrm{y}_i) p(\mathrm{y}_j \mathrm{y}_k) && \text{(eq. 5)} \\ &= p(\mathrm{y}_i) \sum_{\mathrm{y}_k \in \boldsymbol{Z}} p(\mathrm{y}_j \mathrm{y}_k) \\ &= p(\mathrm{y}_i) p(\mathrm{y}_j) && (6) \end{aligned}$$

However, $\mathrm{y}_i \not\perp\!\!\!\perp \mathrm{y}$ thus $\mathrm{y}_i \not\perp\!\!\!\perp \mathrm{y} \implies \mathrm{y}_i \not\perp\!\!\!\perp (\mathrm{y}_j \mathrm{y}_k)$.

**Lemma B.2.**

$$p(\mathrm{y}_i \mid \mathrm{y}_j) > p(\mathrm{y}_j \mid \mathrm{y}_i) \implies p(\mathrm{y}_i \mid \mathrm{y}_j, \boldsymbol{y}_k) > p(\mathrm{y}_j \mid \mathrm{y}_i, \boldsymbol{y}_k)$$

*if* $\mathrm{y}_i, \mathrm{y}_j \perp\!\!\!\perp \boldsymbol{y}_k$

**Proof** We have:

$$\begin{aligned} & p(\mathrm{y}_i \mid \mathrm{y}_j) > p(\mathrm{y}_j \mid \mathrm{y}_i) \\ \implies & p(\mathrm{y}_j) < p(\mathrm{y}_i) && (7) \end{aligned}$$

They were parents and grandparents who thought they had fully grasped the perils facing their teenagers in the tough working-class streets north of Kennedy Airport in Queens . They said that they understood the power of peers , the lure of gangs and drugs , the impact of movies and music and television that with relentless repetition depicted a culture of casual violence . And , they said , they believed they had taken the necessary precautions . They gave their children beepers and cellular phones to keep better track of them . They lined up mentors , and set curfews , and encouraged faith .

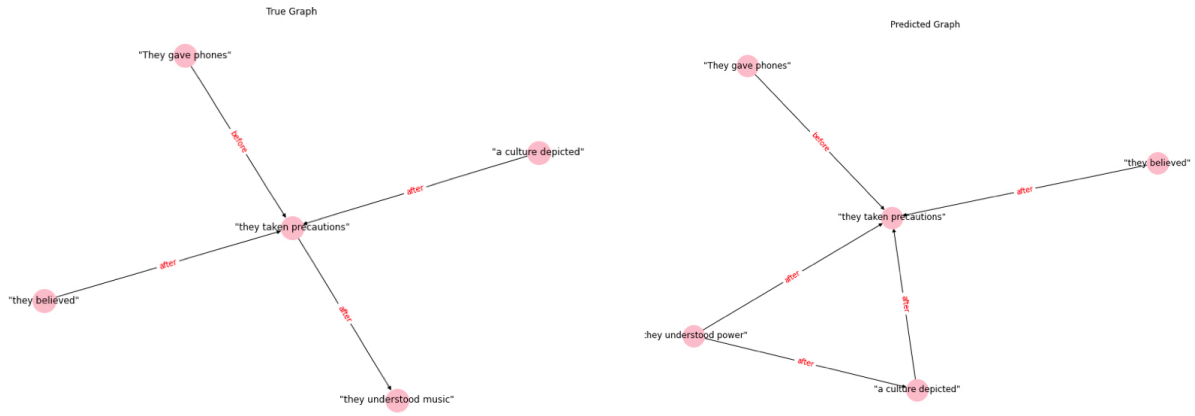



Figure 8

$$
\begin{aligned}
p(\mathrm{y}_j, \boldsymbol{y}_k) &= p(\boldsymbol{y}_k \mid \mathrm{y}_j) p(\mathrm{y}_j) \\
&< p(\boldsymbol{y}_k \mid \mathrm{y}_j) p(\mathrm{y}_i) && \text{(Equation 7)} \\
&= p(\boldsymbol{y}_k \mid \mathrm{y}_i) p(\mathrm{y}_i) && (\mathrm{y}_i, \mathrm{y}_j \perp\!\!\!\perp \boldsymbol{y}_k \implies p(\boldsymbol{y}_k \mid \mathrm{y}_j) = p(\boldsymbol{y}_k \mid \mathrm{y}_i) = p(\boldsymbol{y}_k)) \\
&= p(\mathrm{y}_i, \boldsymbol{y}_k) && (8)
\end{aligned}
$$

Thus,

$$
\begin{aligned}
p(\mathrm{y}_i \mid \mathrm{y}_j, \boldsymbol{y}_k) &= \frac{p(\mathrm{y}_i, \mathrm{y}_j, \boldsymbol{y}_k)}{p(\mathrm{y}_j, \boldsymbol{y}_k)} \\
&> \frac{p(\mathrm{y}_i, \mathrm{y}_j, \boldsymbol{y}_k)}{p(\mathrm{y}_i, \boldsymbol{y}_k)} \\
&= p(\mathrm{y}_j \mid \mathrm{y}_i, \boldsymbol{y}_k) && (9)
\end{aligned}
$$

**Lemma B.3.** *If* $\mathrm{y}_i \perp\!\!\!\perp \mathrm{y}_j \; \forall \mathrm{y}_i, \mathrm{y}_j \in \mathbb{Y}$*, the order is guaranteed to not affect learning.*

**Proof** Let $\pi_j$ be the $j^{th}$ order over $\mathbb{Y}$ (out of $|\mathbb{Y}|!$ possible orders $\Pi$), and $\pi_j(\mathbb{Y})$ be the sequence of elements in $\mathbb{Y}$ arranged with $\pi_j$.

$$
\begin{aligned}
p(\mathrm{y}_i \mid \mathrm{y}_j) &= p(\mathrm{y}_i) && (\mathrm{y}_i \perp\!\!\!\perp \mathrm{y}_j \; \forall \mathrm{y}_i, \mathrm{y}_j) \\
\implies p(\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k) &= p(\mathrm{y}_i) p(\mathrm{y}_j \mid \mathrm{y}_i) p(\mathrm{y}_k \mid \mathrm{y}_i, \mathrm{y}_j) \\
&= p(\mathrm{y}_i) p(\mathrm{y}_j) p(\mathrm{y}_k) \\
\implies p(\pi_m(\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k)) &= p(\pi_n(\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k)) \; \forall \pi_m, \pi_m \in \Pi
\end{aligned}
$$

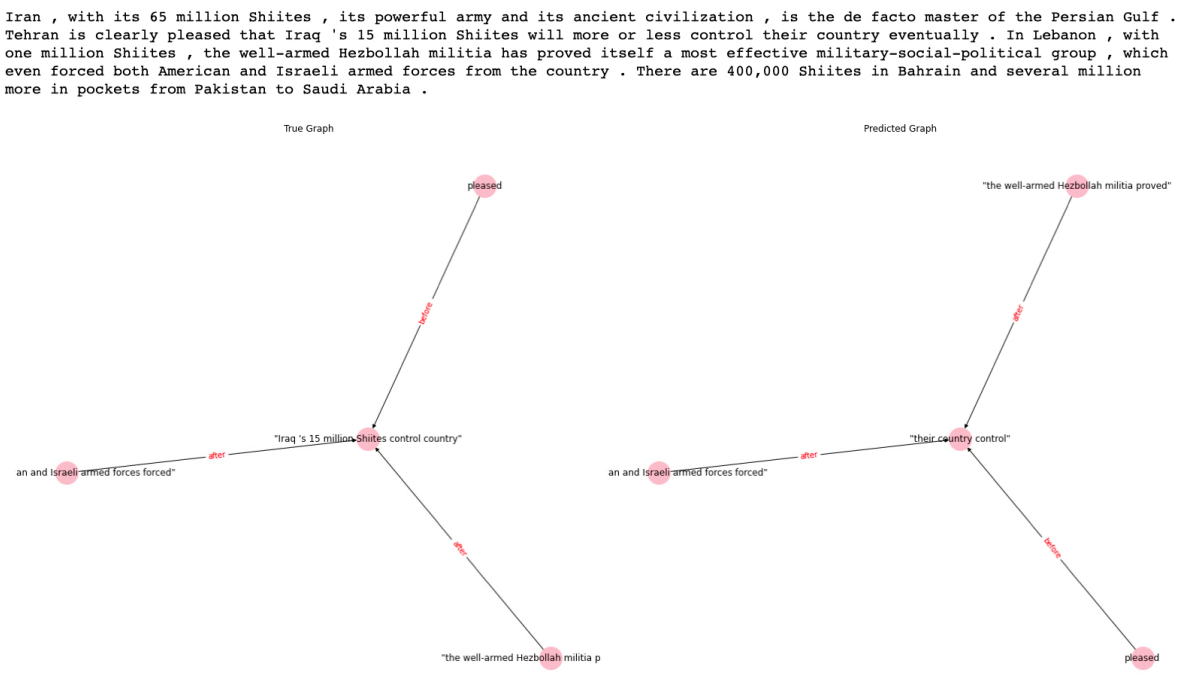



Figure 9

In other words, when all elements are mutually independent, all possible joint factorizations will simply be a product of the marginals, and thus identical.

**Lemma B.4.** *The graphs constructed to sample orders for* TSAMPLE *cannot have cycles.*

**Proof** Let $\mathrm{y}_i, \mathrm{y}_j, \mathrm{y}_k$ form a cycle: $\mathrm{y}_i \to \mathrm{y}_j \to \mathrm{y}_k \to \mathrm{y}_i$. By construction, the following conditions must hold for such a cycle to be present:

$$\log p(\mathrm{y}_j \mid \mathrm{y}_i) - \log p(\mathrm{y}_i \mid \mathrm{y}_j) > \beta \implies \log p(\mathrm{y}_i) < \log p(\mathrm{y}_j)$$
$$\log p(\mathrm{y}_k \mid \mathrm{y}_j) - \log p(\mathrm{y}_j \mid \mathrm{y}_k) > \beta \implies \log p(\mathrm{y}_j) < \log p(\mathrm{y}_k)$$
$$\log p(\mathrm{y}_i \mid \mathrm{y}_k) - \log p(\mathrm{y}_k \mid \mathrm{y}_i) > \beta \implies \log p(\mathrm{y}_k) < \log p(\mathrm{y}_i)$$

Putting the three implications together, we get $\log p(\mathrm{y}_i) < \log p(\mathrm{y}_j) < \log p(\mathrm{y}_k) < \log p(\mathrm{y}_i)$, which is a contradiction. Hence, the graphs constructed for TSAMPLE cannot have a cycle.

## B.2 Sample graphs

In this section, we present additional examples from REUTERS and GO-EMO datasets to illustrate the permutations generated by our method. TSAMPLE encourages highly co-occuring pairs $(\mathrm{y}_i, \mathrm{y}_j)$ to be in the order $\mathrm{y}_i, \mathrm{y}_j$ if $p(\mathrm{y}_j \mid \mathrm{y}_i) > p(\mathrm{y}_i \mid \mathrm{y}_j)$. In our analysis, this dependency in the datasets shows that the orders exhibit a pattern where *specific* labels appear before the *generic* ones. For example, in case of entity typing, the more GO-EMO, *sadness* is generated after the more specific

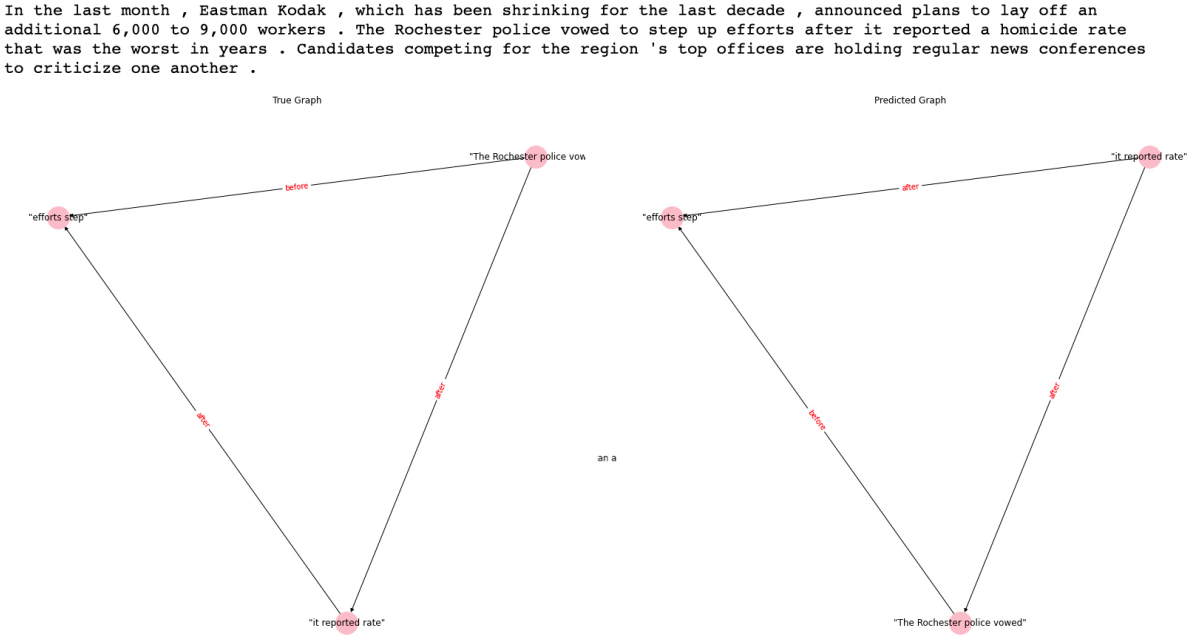



Figure 10

emotion *remorse* and *fear* (Figure 14). Similarly, the entity *crude* is generated after the entities *gas* and *nat-gas*. (Figure 15 (right)).

## B.3 Hyperparameters

We list all the hyperparameters in Table 9.

## B.4 Dataset

Table 10 shows examples for each of the datasets.

He dreams of being a biochemist and revels in science fiction novels , insect collecting and dismantling broken electronic equipment . He acknowledges that his actions were potentially dangerous . '' People ca n't see the difference between showing off , testing the boundaries and killing people , '' he said . '' I went into the school to see what I could get away with , not because I wanted to kill anyone . '' Michael said the most difficult part of his punishment was the six weeks he spent at the Essex County Youth House in Newark . One cellmate , he said , beat him and tried to strangle him .

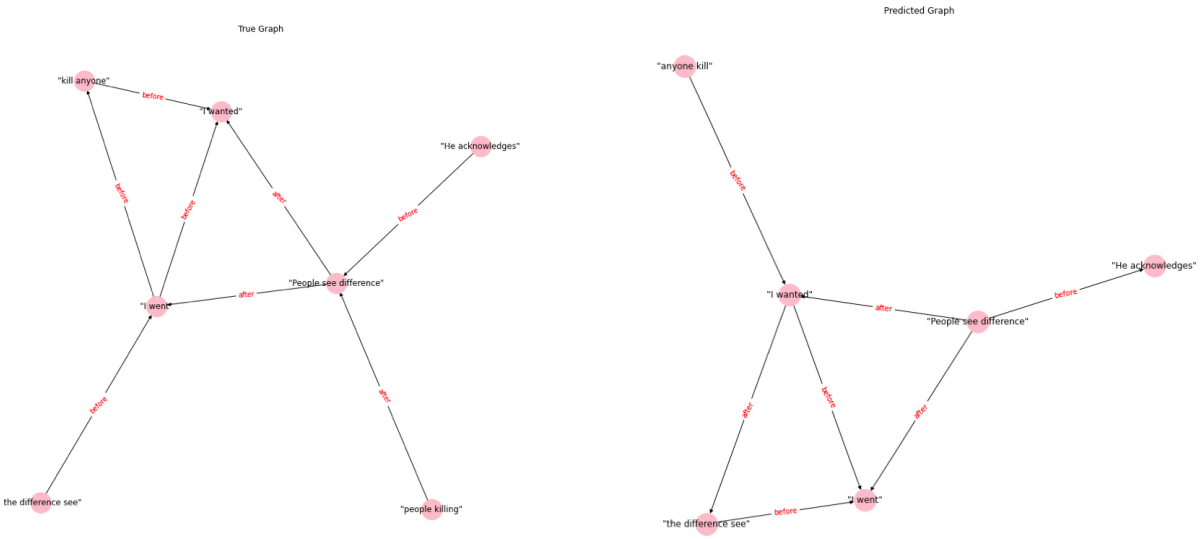



Figure 11

| Hyperparameter | Value |
|---|---|
| GPU | GeForce RTX 2080 Ti |
| gpus | 1 |
| auto_select_gpus | false |
| accumulate_grad_batches | 1 |
| max_epochs | 3 |
| precision | 32 |
| learning_rate | 1e-05 |
| adam_epsilon | 1e-08 |
| num_workers | 16 |
| warmup_prop | 0.1 |
| seeds | [15143, 27122, 999888] |
| add_lr_scheduler | true |
| lr_scheduler | linear |
| max_source_length | 120 |
| max_target_length | 120 |
| val_max_target_length | 120 |
| test_max_target_length | 120 |

Table 9: List of hyperparameters used for all the experiments.

In a dispatch that Mr. Altgens wrote for the agency later that day , he said : `` There was a burst of noise , the second one I heard , and pieces of flesh appeared to fly from President Kennedy 's car . Blood covered the whole left side of his head . Mrs. Kennedy saw what had happened to her husband . She grabbed him , exclaiming , ` Oh , no ! ' '' The Associated Press , in its book on the assassination , `` The Torch is Passed ... '' which was published soon afterward , republished Mr. Altgens 's photograph of the First Lady and the agent with a caption saying it `` shows Secret Service Agent Clint Hill leaping toward Mrs. Kennedy as she desperately moves for help in the first moment of horror . ''

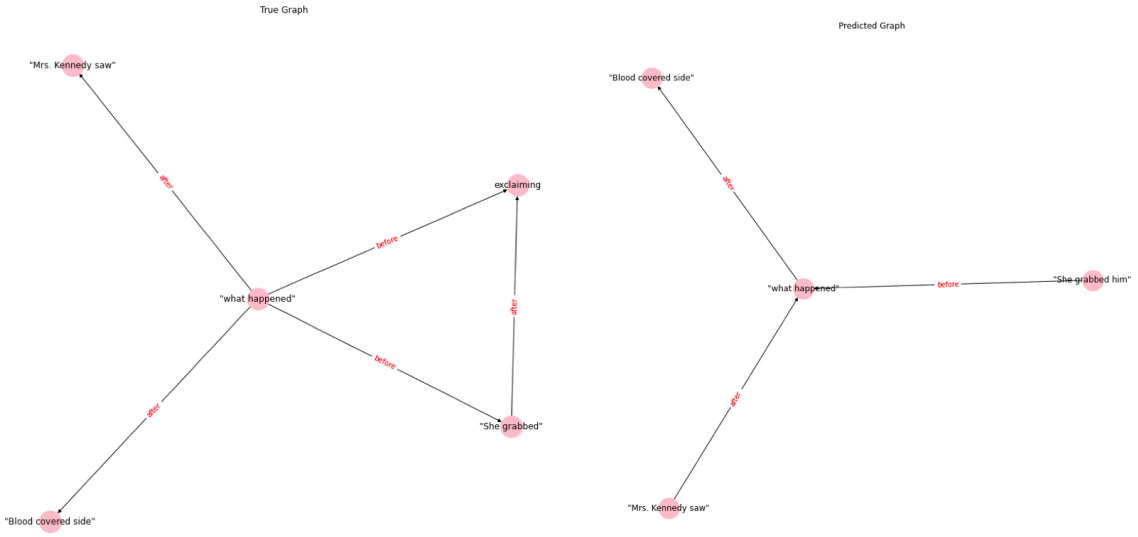



Figure 12

An article yesterday about the release of an Egyptian chemist who had been held by Egyptian police as a suspect in the July 7 London transit bombings misstated the date he was arrested in Cairo . It was July 14 , not June 14 .

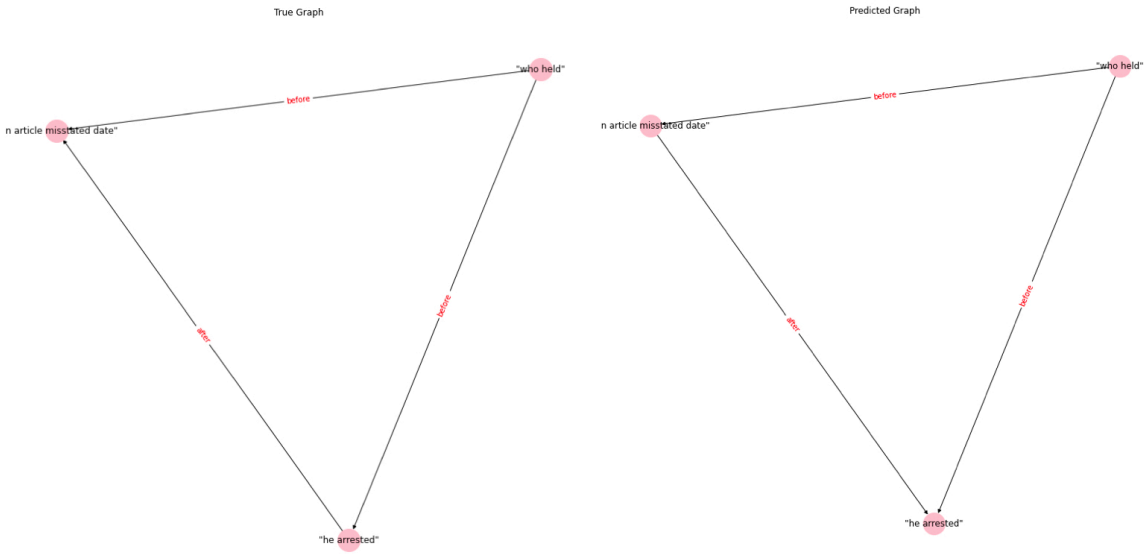



Figure 13

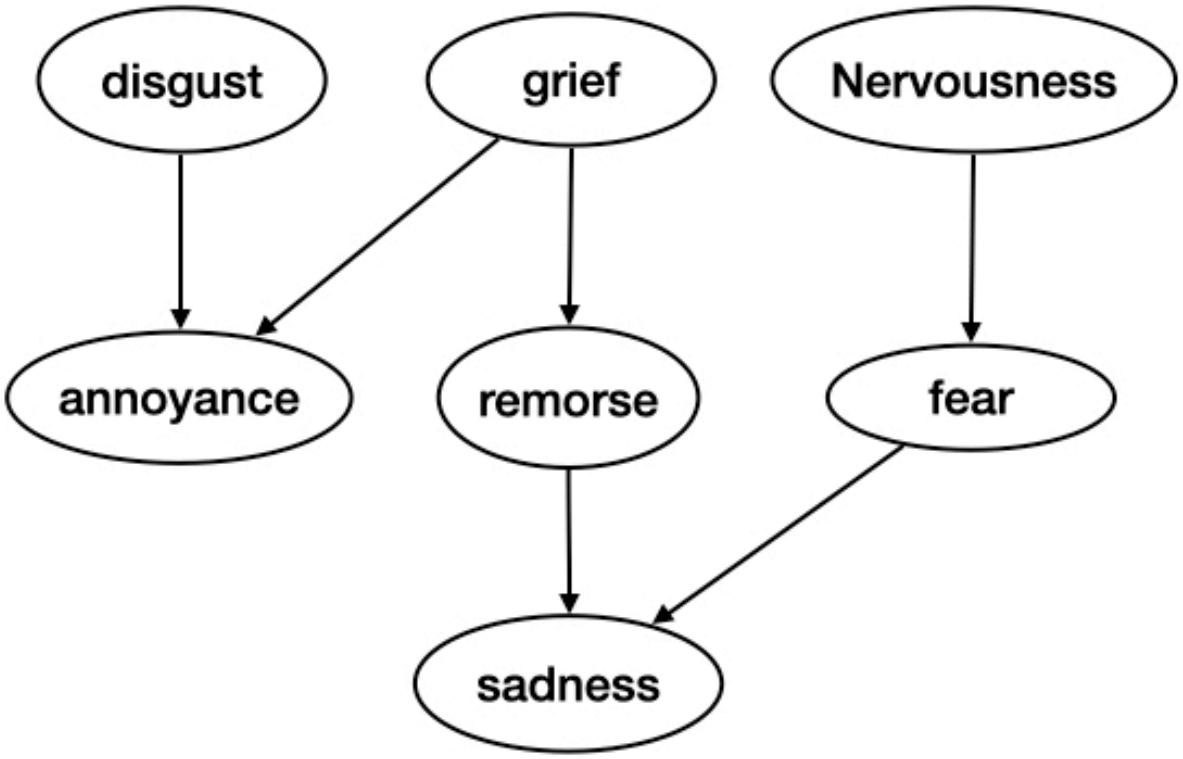



Figure 14: Label dependencies discovered by TSAMPLE for GO-EMO

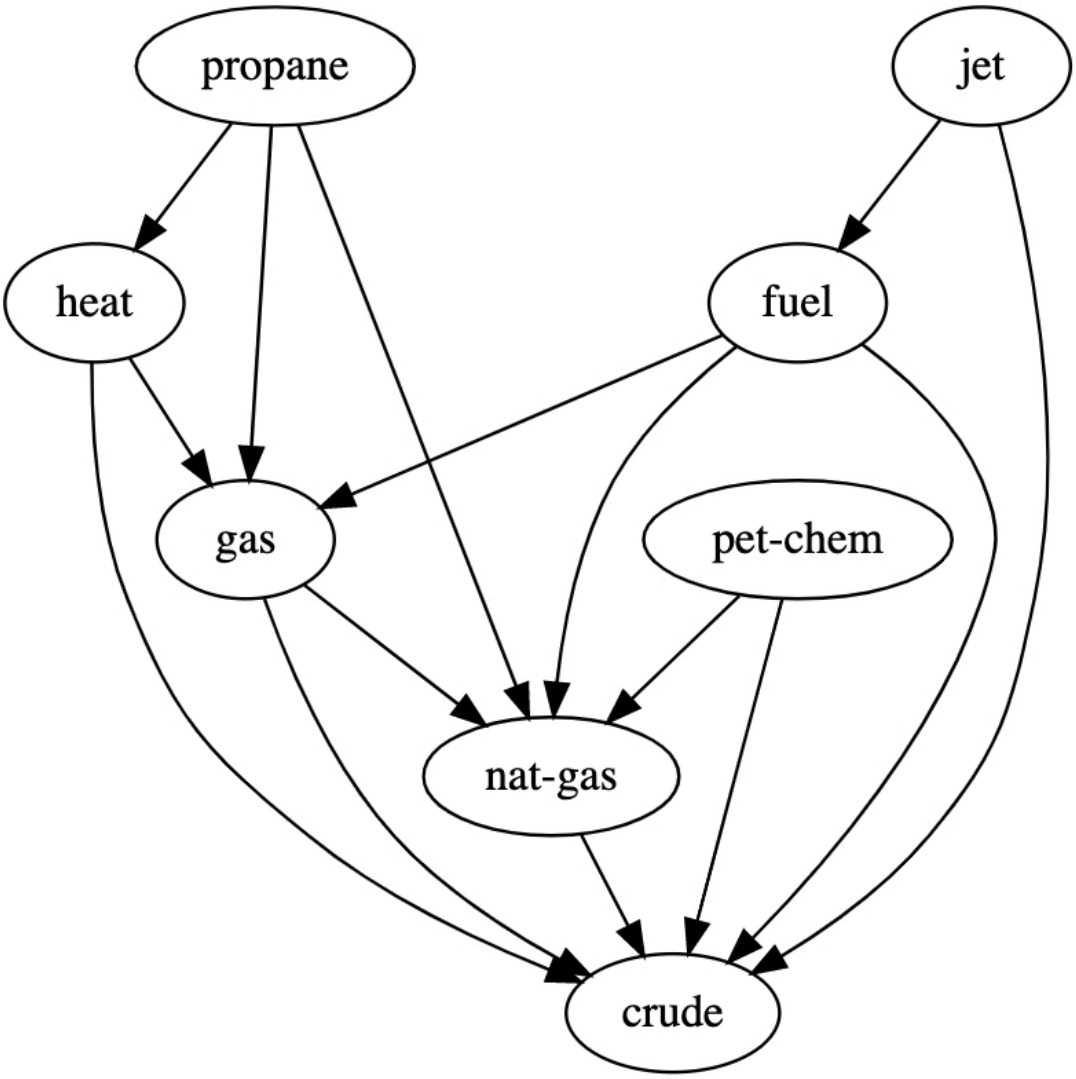



Figure 15: Label dependencies discovered by TSAMPLE for REUTERS

| | Input | Output |
|---|---|---|
| Fine-grained emotion classification, [28] [Demszky et al., 2020] | *So there's hope for the rest of us! Thanks for sharing. What helped you get to where you are?* | {curiosity, gratitude, optimism} |
| Open-entity typing [2519] [Choi et al., 2018] | *Some 700,000 cubic meters of caustic sludge and water burst inundating [SPAN] three west Hungarian villages [SPAN] and spilling.* | {colony, region, location, hamlet, area, village, settlement, community} |
| Reuters [90] [Lewis, 1997] | *India is reported to have bought two white sugar cargoes for. . . . . .cargo sale, they said.* | {ship, sugar} |
| Keyphrase generation [270k] [Ye et al., 2021] | We analyze the impact of core affinity on both network and disk i/o performance...our dynamic core affinity improves the file upload throughput more than digit% | {big data, multi-core, process-scheduling} |

Table 10: Real world tasks used for experiments

| | GO-EMO | OPENENT | REUTERS |
|---|---|---|---|
| MULTI-LABEL | 22.4 | 14.3 | 21.7 |
| MULTI-LABEL @oracle-k | 21.3 | 17.8 | 25.6 |
| TSAMPLE + card | **30.0** | **53.5** | **26.7** |

Table 11: Multi-label classification when the true cardinality is provided to the classifier. While providing the true cardinality helps the performance of multi-label classifiers, it still lags TSAMPLE.

## B.5 Additional results

This section presents detailed results that were omitted from the main paper for brevity. This includes macro and micro precision, recall, and $F$ scores on all datasets, and additional ablation experiments.

1. Table 12 shows the detailed results from the four tasks.
2. Detailed results on GO-EMO, REUTERS, and OPENENT are present in Tables 13, 14, and 15, respectively.
3. Table 16 includes results from a multi-label classification baseline where bert-base-uncased is used as the encoder.

| | GO-EMO | | | OPENENT | | | REUTERS | | | KEYGEN | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $p$ | $r$ | $F$ | $p$ | $r$ | $F$ | $p$ | $r$ | $F$ | $p$ | $r$ | $F$ |
| MULTI-LABEL | 20.8 | 42.4 | 22.4 | 16.4 | 25.1 | 14.3 | 19.7 | 43.4 | 21.7 | - | - | - |
| MULTI-LABEL-K* | 21.3 | 21.3 | 21.3 | 17.8 | 17.8 | 17.8 | 25.6 | 25.6 | 25.6 | - | - | - |
| SET SEARCH | 10.7 | 7.0 | 7.4 | 26.5 | 31.4 | 26.3 | 10.9 | 7.1 | 7.5 | 5.8 | **7.4** | 6.4 |
| SEQ2SEQ | 27.4 | 26.2 | 23.4 | 55.4 | 42.4 | 44.6 | 24.8 | 13.8 | 15.6 | 6.7 | 5.5 | 5.9 |
| RANDOM | 32.5 | 19.9 | 22.7 | 62.6 | 41.7 | 46.9 | 26.7 | 12.7 | 15.2 | 6.6 | 4.5 | 5.2 |
| TSAMPLE | **36.7** | 19.8 | 23.3 | 60.0 | 44.5 | 48.0 | 26.5 | 12.8 | 15.8 | 7.0 | 5.0 | 5.6 |
| SEQ2SEQ + CARD | 33.0 | 28.3 | 26.8 | 62.5 | 44.7 | 50.5 | 34.1 | 21.8 | 24.3 | 7.1 | 5.6 | 6.1 |
| RANDOM + CARD | 35.6 | 26.5 | 27.5 | ***68.6*** | 42.3 | 50.4 | 35.3 | 22.1 | 24.7 | 7.3 | 5.7 | 6.3 |
| TSAMPLE + CARD | 36.1 | **30.5** | **30.0** | *65.5* | ***47.5*** | ***53.5*** | ***36.7*** | ***24.1*** | ***26.7*** | **7.7** | 6.1 | **6.6** |

Table 12: Detailed main results: using permutations generated by TSAMPLE and adding cardinality gives the best overall performance in terms of macro precision, recall, and $F1$- score. MULTI-LABEL is the standard multi-label classification approach. Statistically significant results are *underscored*. CARD stands for cardinality.

| | $p_{\text{micro}}$ | $p_{\text{macro}}$ | $r_{\text{micro}}$ | $r_{\text{macro}}$ | $f_{\text{micro}}$ | $f_{\text{macro}}$ | $J$ |
|---|---|---|---|---|---|---|---|
| SET SEARCH | 47.17 | 10.68 | 13.09 | 7.02 | 10.7 | 7.36 | 7.4 |
| SEQ2SEQ | 41.65 | 27.39 | 35.19 | 26.21 | 27.4 | 23.41 | 23.4 |
| SEQ2SEQ + CARD | 39.77 | 33 | 38.02 | 28.31 | 33 | 26.79 | 26.8 |
| RANDOM + CARD | 44.77 | 35.6 | 32.96 | 26.54 | 35.6 | 27.53 | 27.5 |
| TSAMPLE + CARD | 43.37 | 36.08 | 34.51 | 30.54 | 36.1 | 30.01 | 30 |
| RANDOM- CARD | 48.85 | 32.45 | 27.75 | 19.86 | 32.5 | 22.67 | 22.7 |
| TSAMPLE- CARD | 50 | 36.68 | 29.84 | 19.84 | 36.7 | 23.31 | 23.3 |

Table 13: Results for GO-EMO.

| | $p_{\text{micro}}$ | $p_{\text{macro}}$ | $r_{\text{micro}}$ | $r_{\text{macro}}$ | $f_{\text{micro}}$ | $f_{\text{macro}}$ | $J$ |
|---|---|---|---|---|---|---|---|
| SET SEARCH | 70.04 | 10.92 | 34.9 | 7.1 | 46.56 | 7.54 | 37.49 |
| SEQ2SEQ | 66.36 | 24.74 | 42.28 | 13.78 | 51.64 | 15.58 | 44.3 |
| SEQ2SEQ + CARD | 73.02 | 34.17 | 53.8 | 21.85 | 61.95 | 24.28 | 59.08 |
| RANDOM + CARD | 74.26 | 35.31 | 54.33 | 22.13 | 62.75 | 24.74 | 58.95 |
| TSAMPLE + CARD | 75.65 | 36.67 | 55.54 | 24.13 | 64.05 | 26.66 | 61.14 |
| RANDOM- CARD | 69.56 | 26.68 | 38.15 | 12.71 | 49.27 | 15.2 | 42.24 |
| TSAMPLE- CARD | 76.55 | 26.49 | 41.78 | 12.77 | 54.06 | 15.78 | 47.34 |

Table 14: Results for REUTERS.

| | $p_{\text{micro}}$ | $p_{\text{macro}}$ | $r_{\text{micro}}$ | $r_{\text{macro}}$ | $f_{\text{micro}}$ | $f_{\text{macro}}$ | $J$ |
|---|---|---|---|---|---|---|---|
| SET SEARCH | 24.65 | 26.5 | 29.98 | 31.44 | 23.92 | 26.25 | 13.39 |
| SEQ2SEQ | 52.78 | 55.4 | 39.84 | 42.42 | 41.45 | 44.63 | 24.6 |
| SEQ2SEQ + CARD | 61.26 | 62.48 | 41.87 | 44.68 | 48.07 | 50.48 | 27.84 |
| RANDOM + CARD | 67.56 | 68.59 | 39.61 | 42.25 | 47.98 | 50.4 | 26.89 |
| TSAMPLE + CARD | 64.58 | 65.53 | 44.6 | 47.46 | 51.2 | 53.48 | 29.39 |
| RANDOM- CARD | 60.93 | 62.57 | 39.09 | 41.69 | 44.2 | 46.85 | 25.26 |
| TSAMPLE- CARD | 58.02 | 59.88 | 42.63 | 44.95 | 46.54 | 48.86 | 26.82 |

Table 15: Results for OPENENT.

| | GO-EMO | | | OPENENT | | | REUTERS | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $p$ | $r$ | $F$ | $p$ | $r$ | $F$ | $p$ | $r$ | $F$ | |
| BERT @1 | 31.8 | 10.3 | 15.6 | 38.0 | 10.3 | 15.9 | 31.7 | 12.3 | 17.6 | |
| BERT @3 | 23.8 | 23.4 | 23.6 | 19.7 | 14.0 | 16.1 | 23.4 | 28.3 | 25.5 | |
| BERT @5 | 20.6 | 34.0 | 25.7 | 15.5 | 18.0 | 16.4 | 18.8 | 37.6 | 24.9 | |
| BERT @10 | 16.5 | 54.3 | 25.3 | 11.8 | 26.0 | 16.0 | 15.1 | 61.8 | 24.2 | |
| BERT @20 | 14.1 | 93.2 | 24.5 | 8.4 | 34.3 | 13.5 | 9.5 | 75.9 | 16.8 | |
| BERT @50 | - | - | - | 2.6 | **50.2** | 4.9 | 8.9 | - | - | - |
| BERT | 21.4 | 43.0 | 22.9 | 16.0 | 25.5 | 13.8 | 19.7 | 43.2 | 21.8 | |
| BART @1 | 31.7 | 10.3 | 15.5 | 38.0 | 10.3 | 15.6 | 31.8 | 12.3 | 17.6 | |
| BART @3 | 21.2 | 21.0 | 21.0 | 19.7 | 14.0 | 15.8 | 23.1 | 28.1 | 25.2 | |
| BART @5 | 14.1 | 33.4 | 25.6 | 15.5 | 18.0 | 16.2 | 18.7 | 37.6 | 24.8 | |
| BART @10 | 16.3 | 53.4 | 25.0 | 11.7 | 26.0 | 15.9 | 15.1 | 62.0 | 24.1 | |
| BART @20 | 14.1 | **93.3** | 24.5 | 8.4 | 34.3 | 13.4 | 9.6 | **77.1** | 17.1 | |
| BART @50 | - | - | - | 4.9 | 48.0 | 8.9 | - | - | - | |
| BART | 20.8 | 42.4 | 22.4 | 16.4 | 25.1 | 14.3 | 19.7 | 43.4 | 21.7 | |
| SET SEARCH | 10.7 | 7.0 | 7.4 | 26.5 | 31.4 | 26.3 | 10.9 | 7.1 | 7.5 | |
| SEQ2SEQ | 27.4 | 26.2 | 23.4 | 55.4 | 42.4 | 44.6 | 24.8 | 13.8 | 15.6 | |
| RANDOM | 32.5 | 19.9 | 22.7 | 62.6 | 41.7 | 46.9 | 26.7 | 12.7 | 15.2 | |
| TSAMPLE | **36.7** | 19.8 | 23.3 | 60.0 | 44.5 | 48.0 | 26.5 | 12.8 | 15.8 | |
| SEQ2SEQ +CARD | 33.0 | 28.3 | 26.8 | 62.5 | 44.7 | 50.5 | 34.1 | 21.8 | 24.3 | |
| RANDOM + CARD | 35.6 | 26.5 | 27.5 | ***68.6*** | 42.3 | 50.4 | 35.3 | 22.1 | 24.7 | |
| TSAMPLE + CARD | 36.1 | **30.5** | **30.0** | *65.5* | ***47.5*** | ***53.5*** | ***36.7*** | ***24.1*** | ***26.7*** | |

Table 16: Our main results: using permutations generated by TSAMPLE and adding cardinality gives the best overall performance in terms of macro precision, recall, and $F1$--score score. Statistically significant results are *underscored*. CARD stands for cardinality. BERT @k / BART @k denotes the pointwise classification baseline using BERT/ BART where the top $k$ labels are used as the model output. The average is denoted by BERT/ BART.

### B.6 Fixing the proposal distribution in the VAE formulation

$$
\begin{aligned}
\log p_\theta(\mathbb{Y} \mid \boldsymbol{x}) &= \log \sum_{\pi_{\boldsymbol{z}} \in \Pi} p_\theta(\pi_z(\mathbb{Y}) \mid \boldsymbol{x}) \\
&= \log \sum_{\pi_{\boldsymbol{z}} \in \Pi} \frac{q_\phi(\pi_{\boldsymbol{z}})}{q_\phi(\pi_{\boldsymbol{z}})} p_\theta(\pi_z(\mathbb{Y}) \mid \boldsymbol{x}) \\
&= \log \mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\frac{p_\theta(\pi_z(\mathbb{Y}) \mid \boldsymbol{x})}{q_\phi(\pi_{\boldsymbol{z}})}] \\
&\geq \mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\log p_\theta(\mathbb{Y}, \pi_{\boldsymbol{z}} \mid \boldsymbol{x})] - \mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\log q_\phi(\pi_{\boldsymbol{z}})] \\
\log p_\theta(\mathbb{Y} \mid \boldsymbol{x}) &= \log \sum_{\pi_{\boldsymbol{z}} \in \Pi} p_\theta(\pi_z(\mathbb{Y}) \mid \boldsymbol{x}) \\
&\geq \underbrace{\mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\frac{\log p_\theta(\pi_{\boldsymbol{z}}(\mathbb{Y}) \mid \boldsymbol{x})}{q_\phi(\pi_{\boldsymbol{z}})}]}_{\text{ELBO}} = \mathcal{L}(\theta, \phi)
\end{aligned}
\tag{10}
$$

Where equation 10 is the evidence lower bound (ELBO). The success of this formulation depends on the quality of the proposal distribution $q$ from which the orders are drawn. When $q$ is fixed (e.g., to uniform distribution over the orders), learning only happens for $\theta$. This can be clearly seen from splitting equation 10 into terms that involve just $\theta$ and $\phi$:

$$
\begin{aligned}
\nabla_\phi \mathcal{L}(\theta, \phi) &= 0 \\
\nabla_\theta \mathcal{L}(\theta, \phi) &= \nabla_\theta \mathbb{E}_{q_\phi(\pi_{\boldsymbol{z}})}[\log p_\theta(\mathbb{Y}, \pi_{\boldsymbol{z}} \mid \boldsymbol{x})]
\end{aligned}
$$

# C Chapter 4: Learning Performance Improving Code Edits

# D Analysis of Generated Code Edits

**Algorithmic Transformations** ($34.15\%$)**.** The most dominant transformation, representing approximately $34.15\%$ of the changes, is the *Algorithmic* category. Edits in this category exhibited sophisticated code restructuring. A frequent transformation was the shift from recursive methodologies to dynamic programming approaches, which can significantly enhance running time for specific problem types. Other examples include replacing Binary Indexed Trees with more straightforward constructs, removing redundant conditional checks, bit manipulations, and in some cases, using identities from number theory and algebra to replace complex computation with a formula.

**Input/Output Operations** ($26.02\%$)**.** The *Input/Output operations* category, accounting for roughly $26.02\%$ of the changes, primarily centered on transitioning from C++ standard I/O methods (‘cin/cout‘) to the faster C-standard methods (‘scanf/printf‘). Other examples include reading a string character-by-character vs. reading in one go, This transformation is particularly beneficial for problems dealing with extensive datasets, where I/O operations can be a bottleneck.

**Data Structure Modifications (21.14%).** Changes in the *Data Structures* category, which constituted about 21.14% of the transformations, showcased the model's adeptness in selecting optimal data structures for the task. A recurring modification was the transition from vectors to traditional arrays, leading to enhanced access times and reduced overhead. Additionally, the changes include removal of pointers in favor of direct access, and using hashmaps when appropriate.

**Miscellaneous Optimizations (18.70%).** The *Miscellaneous* category, encompassing approximately 18.70% of changes, captured a myriad of optimizations. These ranged from code cleanups, such as omitting unnecessary initializations, to replacing computationally intensive functions with predefined constants.

While our analysis showcases a variety of optimizations, it is essential to address certain speedup sources that may be considered spurious. Specifically, in 10 out of the 120 cases we examined, the speedup stemmed from reducing the constants used to allocate arrays. These speedups might not always reflect genuine algorithmic improvements, and indicate that the test cases may not perfectly cover all the cases, an open problem in code synthesis [Li et al., 2021b]. Thus, while they contribute to the overall speedup metrics, they should be interpreted with caution. Nevertheless, our analysis shows that the vast majority of speedups do not suffer from this issue, supporting our strong empirical results.

## D.1 Examples of Optimizations

We show several examples to demonstrate the nature of optimizations made by our model. In these examples, we highlight the removal of a wasteful nested loop (Figure 17), eliminating the need to sort (Figure 16), avoiding unnecessary precomputations (Figure 18), and use of simple modular arithmetic properties for optimization (Figure 19).

```cpp
int main(){
    int n, m, a, b;
    vector<int> v, v1;

    cin >> n >> m;

    for(int i = 0; i < m; i++){
        cin >> a >> b;
        v.push_back(a);
        v1.push_back(b);
    }

    sort(v.begin(), v.end());
    sort(v1.begin(), v1.end());

    if(v.back() > v1[0]){
        cout << 0 << endl;
    } else {
        cout << v1[0] - v.back() + 1 <<
    endl;
    }

    return 0;
}
```

(a) Slower Code.

```cpp
int main(){
    int n, m, a, b, max = -1, min = 1e9;
    scanf("%d%d", &n, &m);
    for(int i = 0; i < m; i++){
        scanf("%d%d", &a, &b);
        if(a > max) max = a;
        if(b < min) min = b;
    }
    ans = min - max + 1;
    if(ans < 0){
        ans = 0;
    }
    printf("%d\n", ans);
    return 0;

}
```

(b) Faster Code.

Figure 16: Comparison of two programs for determining the range between the maximum and minimum values from a set of input pairs. The faster code (right) generated by GPT-2 directly computes the maximum start and minimum end of the ranges in a single pass ($\mathcal{O}(n)$), eliminating the need for sorting ($\mathcal{O}(n \log n)$).

```cpp
int main(){
   int k,x;
   cin>>k>>x;
    for (int i=-1000000;i<1000001;i++) {
    if(i==x){
       for (i=x-(k-1);i<x+k;i++){
            cout<< i<<" ";
        }
    }
   }
   return 0;
}
```

(a) Slower Code.

```cpp
int main(){
    int k,x;
    scanf("%d %d",&k,&x);
    for(int i=x-k+1;i<=x+k-1;i++)
        printf("%d ",i);
    return 0;
}
```

(b) Faster Code.

Figure 17: Comparison of two code implementations for printing $2k-1$ consecutive numbers centered around the input $x$. The faster code (right) optimizes the process by directly computing the range without the need for nested loops, resulting in a more efficient and concise solution. The red highlighted portion in the slower code (left) indicates the wasteful nested loop that was eliminated in the optimized version. This loop unnecessarily iterates over a large range of numbers, only to perform a meaningful operation for a tiny fraction of those iterations.

```cpp
int main()
{
    int i, n;
    long long num[100005] = {0,1};
    for (i = 2; i <= 100004; i++)
        num[i] = (num[i-1] *
    i)%(1000000007);
    scanf("%d", &n);
    printf("%lld\n", num[n]);
    return 0;
}
```

(a) Slower Code.

```cpp
long long a=1,mod=1e9+7;
int n;
int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        a=(a*i)%mod;
    }
    printf("%lld",a);
}
```

(b) Faster Code.

Figure 18: Comparison of two code implementations for computing factorial modulo $10^9 + 7$. The slower code (left) precomputes the factorial for all numbers up to $10^5$, storing them in an array. The faster code (right) computes the factorial only for the given input, resulting in a more memory-efficient and faster solution. The red highlighted portion in the slower code indicates the precomputation step that was eliminated in the optimized version.

```cpp
int main() {
    int A, B, C;
    scanf("%d %d %d", &A, &B, &C);

    bool isYes = false;
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            if ((A * i) - (B * j) == C)
                isYes = true;
        }
    }

    printf("%s\n", isYes ? "YES" : "NO");
    return 0;
}
```

(a) Slower Code with Nested Loops.

```cpp
int main() {
    int A, B, C;
    scanf("%d %d %d", &A, &B, &C);

    bool is_yes = false;
    for (int i = 0; i < B; i++) {
        if ((A * i) % B == C)
            is_yes = true;
    }

    printf("%s\n", is_yes ? "YES" : "NO");
    return 0;
}
```

(b) Optimized Code.

Figure 19: Optimization of a modular arithmetic problem. The slower code naively checks all possible combinations of `i` and `j` leading to a complexity of $\mathcal{O}(10^6)$. The faster code leverages the property of modular arithmetic, reducing the complexity to $\mathcal{O}(B)$. By directly computing the modulo operation for each `i` in the range $[0, B-1]$, it efficiently determines if the condition $(A \times i) \mod B = C$ is satisfied. Note that the example on the right is faster, but the generated code could have been even faster if it included a break statement.

## D.2 Convergence of GPT3.5 Fine-Tuned Models with Additional Generations

The data in Section 4.3.4 shows that the gap between GPT-3 fine-tuned on HQ data and HQ + Self-Play seems to diminish with more generations. Training with HQ data helps increase the model's coverage, allowing it to optimize a large number of programs with just a single greedy sample. However, as more samples are drawn, the performance of HQ and HQ + Self-play gradually converge to a similar level of performance. We include the plots of the performance improvements as the number of samples gradually increases in Figure 20 and Figure 21.

Additionally, there is a slight drop in correctness after training with Self-Play; however, the speedup and correctness increase from 6.74 → 6.86 and 86.66 → 87.68. This reveals a precision-recall style trade-off: the model trained on synthetic data learns to try novel optimization strategies, but that comes at the cost of making more mistakes. We will add this analysis to the revision.

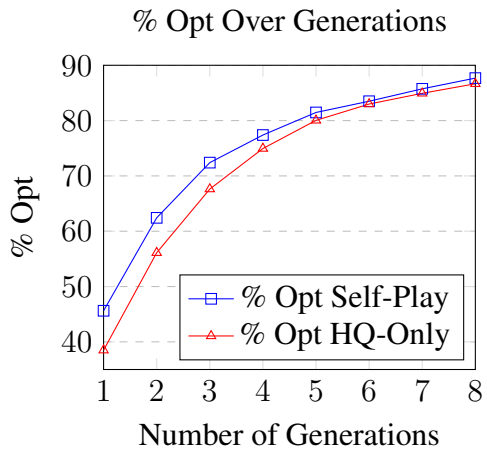



Figure 20: Performance in % Opt over generations comparison of GPT-3 fine-tuned with HQ Data Only vs. HQ + Self-Play.

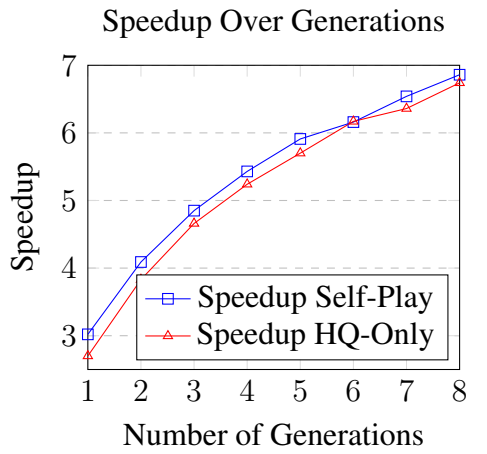



Figure 21: Speedup over generations comparison of GPT-3 fine-tuned with HQ Data Only vs. HQ + Self-Play.

Table 17: Error analysis of GPT-3.5 fine-tuned with synthetic data.

| **Result** | **Percentage** |
|---|---|
| Failed to compile (syntax/type errors) | 12.51% |
| Compiled, but got [95-100%] of test cases wrong | 27.59% |
| Compiled, but got (75, 95%] of test cases wrong | 12.08% |
| Compiled, but got (25, 75%] of test cases wrong | 10.84% |
| Compiled, but got (0-25%] of test cases wrong (at least 1 test case was wrong) | 6.90% |
| Ran all test cases, but the program was slower than the original | 9.93% |
| Ran all test cases, but the program was the same speed as the original | 9.40% |
| Ran all test cases, the program was faster, but not 1.1× speedup or higher | 10.75% |

## D.3 Error Analysis

We performed error analysis on the the GPT-3.5 fine-tuned with Self-Play. We analyzed the generated programs that it fails to optimize and the cause of each failure. Table 17 shows that a large fraction $\sim$60% of the failures happen because the proposed changes break a unit test. In about 30% of the cases, the model produces a correct, but the generated program is either slower (10%) or doesn't meet our threshold for speedup (10%). Finally, in about 10% of the cases, the generated program has a syntax error. Additionally, with test cases, we find that when the model gets the program wrong, it seems to most often get it *quite wrong* by missing most test cases.

Additionally, we conduct additional analysis to investigate the properties of programs that PIE fails to optimize. The results show a mild negative correlation between the problem description length and average accuracy (-0.15) and between the source program length and average accuracy (-0.26), suggesting longer inputs slightly reduce accuracy. Additionally, the average speedup has a mild negative correlation with both the problem description length (-0.16) and the source program length (-0.11), indicating a minimal impact of length on speedup compared to correctness. Overall, this analysis reveals that language models struggle to generate a correct program when faced with larger source programs and challenging problems, but their ability to optimize programs is minimally impacted. This motivates a future work direction where techniques from program repair may be combined with PIE for better results.

**Why does Performance-Conditioning Degrade the Ability to Produce Correct Code?** We believe that conditioning the model only to generate programs with a 10/10 optimization rate may constrain the number of optimizations available for any given input. To investigate this, we experimented using the first 6 generations from the 7b Performance-Conditioned model when conditioned on 10/10 versus combining the first 2 generations when conditioned on 10/10, 9/10, and 8/10 (i.e. comparing 6 total generations from one strategy vs. 6 total generations across difference strategies). When we did this, we saw a %Correct increase from 59.95% to 64.36%. These results support the explanation that performance labels may restrict the set of generated programs that are correct.

## D.4 PIE Dataset Details

| Dataset | Unique Problem IDs |
|---|---|
| Train | 1,474 |
| Val | 77 |
| Test | 41 |

Table 18: Number of unique problem ids.

| Dataset | Pairs |
|---|---|
| Train | 77,967 |
| Val | 2,544 |
| Test | 978 |

Table 19: Number of pairs.

| Dataset | Mean src | Mean tgt | Median src | Median tgt |
|---|---|---|---|---|
| Train | 675.00 | 616.44 | 417 | 372 |
| Val | 644.74 | 471.47 | 180 | 110 |
| Test | 427.85 | 399.15 | 362 | 319 |

Table 20: GPT-2 Tokenizer lengths.

## D.5 Self-Play Data Generation Details

We use the template in Figure 22 for prompting GPT-3 in the self-play scenario. For the prompt, we sample natural language descriptions of programming problems as well as accepted solutions to fill in the template. For generation, we use a temperature of 1.0 and use top-p sampling with $p = 0.9$ For each prompt, we try attempt to take $n = 5$ samples. We chose these samples after doing a sweep of 6 configurations of generation parameters, each attempting to generate 200 programs. We found this configuration to be the most cost-effective per new-sample with relatively promising rates of novelty.

We found that after attempting to generate 10,000 new programs through the prompting strategy, 6,553 were not in the training/validation/test set of PIE. We keep track of equivalent programs of the ones generated, and of these 6,553 generations we found 3,314 equivalence sets. In total, this required executing over 1.4 million binary input pairs. Parallelized on a 24-core Intel 13900k processor with 64GB of RAM, this took less than 72 hours to complete.

```
Description 1: {description_1}
Code 1: {code_1}
Description 2: {description_2}
Code 2: {code_2}
Now, can you generate a program that takes that same input as Code
↪  2 in Code 3 but produces different outputs? Write it to be as
↪  novel as possible.
Code 3:
```

Figure 22: The prompt template used for prompting GPT-3 for generating synthetic data for self-play.

## D.6 Ablation of Retrieval-Based Few-Shot Prompting Configuration

For our retrieval-based prompting experiment we tried multiple configurations for the number of retrieved prompts where of $K = \{1, 2, 4\}$ of the $K$ closest retrieved prompts.

Table 21: Retrieval-based few-shot prompting ablation over different $K$ examples for retrieval and over various models.

| Method | Model | Best@1 | | | Best@8 | | |
|---|---|---|---|---|---|---|---|
| | | %Opt | Speedup | %Correct | %Opt | Speedup | %Correct |
| Dynamic Retrieval, K=1 | CODELLAMA 7B | 3.27% | 1.09× | 16.67% | 15.64% | 1.50× | 50.51% |
| Dynamic Retrieval, K=1 | CODELLAMA 13B | 5.32% | 1.16× | 21.68% | 22.29% | 1.72× | 62.99% |
| Dynamic Retrieval, K=1 | CODELLAMA 34B | 10.02% | 1.25× | 30.67% | 34.25% | 2.21× | 69.73% |
| Dynamic Retrieval, K=2 | CODELLAMA 7B | 4.40% | 1.13× | 20.55% | 16.87% | 1.51× | 55.32% |
| Dynamic Retrieval, K=2 | CODELLAMA 13B | 9.10% | 1.35× | 28.73% | 28.02% | 1.97× | 64.72% |
| Dynamic Retrieval, K=2 | CODELLAMA 34B | 10.22% | 1.27× | 25.87% | 34.25% | 2.28× | 63.19% |
| Dynamic Retrieval, K=4 | CODELLAMA 7B | 6.34% | 1.19× | 23.11% | 21.06% | 1.66× | 57.98% |
| Dynamic Retrieval, K=4 | CODELLAMA 13B | 9.30% | 1.29× | 26.99% | 28.12% | 2.04× | 62.58% |
| Dynamic Retrieval, K=4 | CODELLAMA 34B | 11.66% | 1.34× | 30.57% | 42.54% | 2.43× | 73.62% |
| Dynamic Retrieval, K=2 | GPT3.5 | 26.18% | 1.58× | 80.37% | 48.06% | 2.14× | **97.85%** |
| Dynamic Retrieval, K=2 | GPT-4 | **50.00%** | **2.61**× | **80.57%** | **74.74%** | **3.95**× | **97.85%** |

## D.7 Training Details

We fine-tuned the 7B and 13B variants using the HuggingFace Transformers library with FSDP to distribute the training process across 8× 48GB GPUs (NVIDIA RTX A6000/NVIDIA L40). For our high-quality dataset, which consists of approximately 4,000 examples, the models were fine-tuned until convergence was achieved, which can be done under 12 hours with 8 GPUs. For tasks related to full data fine-tuning and performance-conditioned fine-tuning, we only train for

1 epoch, which takes 24 to 36 hours, depending on the model of GPU used. All experiments were conducted using the AdamW optimizer [Loshchilov and Hutter, 2017]. For the 7B and 13B variants of CODELLAMA, we used a batch size of 32 and a learning rate of $1e{-5}$ for all of the experiments.

## D.8 Example of Duplicate Code in CodeNet with Different Measured Runtimes

Figure 23 contains an example of code we found duplicated across the Project Codenet Dataset with variance in the dataset's report of `CPUTime`. For problem number `p03160` and between submission `s766827701` and `s964782197` a speedup of 2.44× is reported, despite the programs and environments being identical. We note that multiple submissions existed, because it was template code. For brevity, we remove the macros, imports, and comments.

## D.9 Lora Results

We show results using low-rank adaptors for finetuning in Table 22. We hypothesize that this gap may be because performance optimization examples do not occur naturally in the training data.

Recent work has shown that the effectiveness of parameter-efficient methods depends on the training data. For example, He et al. [2021] find that "PEFT techniques are slower to converge than full tuning in low/medium-resource scenarios," and Niederfahrenhorst et al. [2023] find that LoRA is least effective for challenging tasks like mathematical reasoning. Together, these works indicate that the performance of PEFT may be heavily task-dependent. Our hypothesis is based on the fact that LoRA only changes a small subset of the model's parameters, and is likely most helpful when the base model has some proficiency for the task (due to pre-training), and LoRA can help adapt the model of the task further. Given that LLMs generally struggled in program optimization without retrieval or full fine-tuning, we hypothesize that the challenging nature of the problem and a potential lack of pre-trained proficiency pose challenges for LoRA.

Table 22: LoRA Experiments: Results for fine-tuning CODELLAMA with low rank adapters. A LoRA rank of 32 and LoRA alpha of 16 is used for all experiments listed.

| | | **Best@1** | | | **Best@8** | | |
|---|---|---|---|---|---|---|---|
| **Dataset** | **Model** | %Opt | Speedup | %Correct | %Opt | Speedup | %Correct |
| All | CODELLAMA 7B | **1.12**% | 1.01× | 45.82% | 9.57% | **1.17**× | 87.47% |
| All | CODELLAMA 13B | 0.41% | 1.01× | 59.47% | 9.67% | 1.15× | 90.94% |
| HQ | CODELLAMA 13B | 0.92% | **1.02**× | **59.57**% | **10.69**% | **1.17**× | **91.04**% |

## D.10 Prompts

```
using namespace std;
typedef long long ll;
inline void getInt(int* p);
const int maxn=1000010;
const int inf=0x3f3f3f3f;
ll n;
ll dp[maxn];
ll a[maxn];
int main()
{
    gbtb;
    cin>>n;
    repd(i,1,n)
    {
        cin>>a[i];
    }
    dp[1]=0;
    dp[0]=0;
    dp[2]=abs(a[2]-a[1]);
    repd(i,3,n)
    {
        dp[i]=min(dp[i-2]+abs(a[i]-a[i-2]),dp[i-1]+abs(a[i]-a[i-1]));

    }
    cout<<dp[n];
    return 0;
}

inline void getInt(int* p) {
    char ch;
    do {
        ch = getchar();
    } while (ch == ' '  ch == '\n');
    if (ch == '-') {
        *p = -(getchar() - '0');
        while ((ch = getchar()) >= '0' && ch <= '9') {
            *p = *p * 10 - ch + '0';
        }
    }
    else {
        *p = ch - '0';
        while ((ch = getchar()) >= '0' && ch <= '9') {
            *p = *p * 10 + ch - '0';
        }
    }
}
```

Figure 23: An example of a C++ program we found multiple submissions for as it is template code. Across these submissions, we found variance in the reported CPU runtime despite the code and competitive programming environment being identical.

# E Chapter 6: Think about it! Improving defeasible reasoning by first modeling the question scenario

## E.1 Training graph corrector

As mentioned in Section 6.3.2, the graph generator $\text{GEN}_{\text{init}}$ is trained as a seq2seq model from WIQA with `input` = [Premise] $\mathbf{T}_i$ | [Situation] $\mathbf{S}_i$ | [Hypothesis] $\mathbf{H}_i$, and `output` = $\mathbf{G}_i$. Graphs in WIQA additionally capture the influence that the situation has on the hypothesis.

```
Given the program below, improve its performance:

### Program:
{src_code}

### Optimized Version:
```

Figure 24: Instruction-prompting for adapting LLMs. The model is provided with direct instructions to improve the performance of the given program.

```
slow1 → fast1 || slow2 → fast2 || slow3 → fast3 || ... || slowN →
↪  fastN

### Program:
{src_code}

### Optimized Version:
```

Figure 25: Few-shot prompting for in-context learning. The format "slow $\rightarrow$ fast" is used for adaptation. A test program is appended for inference.

Denoting this influence label by $y_i$ can be either *helps* or *hurts*

From our experiments, we observe that appending $y_i$ to the training data (from `input` = [Premise] $\mathbf{T}_i$ | [Situation] $\mathbf{S}_i$ | [Hypothesis] $\mathbf{H}_i$ to `input` = [Premise] $\mathbf{T}_i$ | [Situation] $\mathbf{S}_i$ | [Hypothesis] $\mathbf{H}_i$ | $y_i$) reduces repetitions by 13%.

We refer to this data generator as $\text{GEN}^*_{\text{init}}$, and the graphs produced by it as $\mathbf{G}^*$. However, we do not have access to $y$ during test time, and thus $\text{GEN}^*_{\text{init}}$ cannot be used directly to produce $\mathbf{G}^*$ for defeasible queries. We circumvent this by using $\text{GEN}^*_{\text{init}}$ to train a graph-to-graph generation model, that takes as input $\mathbf{G}'$ and generates $\mathbf{G}^*$ as output ($\mathbf{G}' \rightarrow \mathbf{G}^*$). We call this system $\text{GEN}_{\text{corr}}$. We give an overview of the process in Figure 30. In Figure 31, we give examples of an intial graph produced by $\text{GEN}_{\text{init}}$, the corresponding graph produced by $\text{GEN}^*_{\text{init}}$, and the graph produced by $\text{GEN}_{\text{corr}}$.

## E.2 MOE gradient analysis

We restate Equation 6.2 for quick reference:

$$\mathbf{p} = \mathbf{M}(\mathbf{x})$$
$$\mathbf{o} = \sum_{i=1}^{n} p_i \mathbf{E_i}(\mathbf{x})$$

where we have changed the notation slightly to use $\mathbf{o}$ as the MOE output instead of $\mathbf{y}$. We also refer to $\mathbf{E_i}(x)$ as $\mathbf{E_i}$. Further, $o_j = \sum_{i=1}^{n} p_i E_{ij}$. We present the analysis for a generic multi-class

```
### slower program:
{src_code}
Give a potential strategy improving the program:
```

LLM reasoning: The current program calculates the sum of integers from 1 to n-1 using a loop. This iterative approach results in a time complexity of O(n). The sum of the first n natural numbers follows an arithmetic progression, and by leveraging Gauss's formula, n(n-1)/2, we can compute the sum directly. This formula-based approach allows us to calculate the sum in constant time, O(1), eliminating the need for iteration.

```
### optimized version of the same code with the strategy mentioned
↪  above:
```

LLM Response: optimized code

Figure 26: Chain-of-thought prompting. The model's intermediate response and final program are highlighted in blue, indicating they are produced by the LLM.

```
similar_slow1 → similar_fast1 || similar_slow2 → similar_fast2 ||
↪  ... || similar_slowN → similar_fastN

### Program:
{src_code}

### Optimized Version:
```

Figure 27: Retrieval-based few-shot prompting. By dynamically retrieving analogous program structures or challenges, the model is guided to better harness patterns in PIE.

classification setting with $k$ classes, with training done using a cross-entropy loss $\mathcal{L}$ (Figure 32) Let $\hat{y}_c$ be the normalized probability of the correct class $c$ calculated using softmax:

$$\begin{aligned}\hat{y}_c &= \frac{\exp(o_c)}{\sum_{j=1}^{k}\exp(o_j)} \\ &= \frac{\exp(\sum_{i=1}^{n} p_i E_{ic})}{\sum_{j=1}^{k}\exp(\sum_{i=1}^{n} p_i E_{ij})}\end{aligned}$$

Let $\mathcal{L}$ be the cross-entropy loss:

```
Below is a program. Optimize the
↪  program and provide a more
↪  efficient version.

### Program:
{src_code}

### Optimized Version:
{tgt_code}
```

(a) Training Prompt.

```
Below is a program. Optimize
↪  the program and provide a
↪  more efficient version.

### Program:
{src_code}

### Optimized Version:
```

(b) Inference Prompt.

Figure 28: Training and inference prompts for unconditional optimization with GPT-2.

$$\mathcal{L} = -\log \hat{y}_c = -o_c + \log \sum_{j=1}^{k} \exp(o_j)$$
$$= -\sum_{i=1}^{n} p_i E_{ic} + \log \sum_{j=1}^{k} \exp(\sum_{i=1}^{n} p_i E_{ij})$$

**Evaluating $\frac{\partial \mathcal{L}}{\partial p_m}$** The derivatives w.r.t. the $m^{th}$ expert gate probability $p_m$ is given by:

$$\frac{\partial \mathcal{L}}{\partial p_m} = -E_{mc} + \frac{\sum_{j=1}^{k} E_{mj} \exp(\sum_{i=1}^{n} p_i E_{ij})}{\sum_{j=1}^{k} \exp(\sum_{i=1}^{n} p_i E_{ij})}$$
$$= -E_{mc} + \sum_{j=1}^{k} \hat{y}_j E_{mj}$$
$$= -E_{mc}(1 - \hat{y}_c) + \sum_{j=1, j \neq c}^{k} \hat{y}_j E_{mj} \tag{11}$$

**Evaluating $\frac{\partial \mathcal{L}}{\partial E_{mc}}$** the derivatives w.r.t. the logits $E_{mc}$ (logit for the correct class by $m^{th}$ expert) is given by:

$$\frac{\partial \mathcal{L}}{\partial E_{mc}} = -p_m + \frac{\exp(o_c) p_m}{\sum_{j=1}^{k} \exp o_j}$$
$$= -p_m(1 - \hat{y}_c) \tag{12}$$

Equations 11 and 12 have natural interpretations: the gradient on both the mixture probability $p_m$ and the logits $E_{mc}$ will be 0 (note that for Equation 11, $\mathbf{y}^c = 1 \implies \mathbf{y}^j = 0$ for $j \neq c$) when the

```
// Retrieved 1-nearest prompt, slower
    src_code
#include <iostream>
#include <stack>
using namespace std;

stack<char> s;

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        char t;
        cin >> t;
        if (s.empty())
            s.push(t);
        else if (t == s.top())
            ;
        else
            s.push(t);
    }
    cout << s.size();
    return 0;
}
```

(a) Retrieved Slow.

```
// Retrieved 1-nearest prompt, faster
    tgt_code
#include <cstdio>

int n, ans;
char ch1, ch2;

int main() {
    scanf("%d", &n);
    ch1 = getchar();
    ch1 = getchar();
    ans = 1;
    for (int i = 1; i < n; i++) {
        ch2 = getchar();
        if (ch2 != ch1) ans++;
        ch1 = ch2;
    }
    printf("%d", ans);
}
```

(b) Retrieved Fast.

```
// Code for to_be_optimized goes here
#include <bits/stdc++.h>
using namespace std;

int main() {
    int N, len;
    cin >> N;
    string s;
    cin >> s;
    len = s.size();
    if (len > N) {
        for (int i = len; i > N; i--) {
            s.pop_back();
        }
        for (int j = 0; j < 3; j++) {
            s.push_back('.');
        }
        cout << s;
    } else {
        cout << s;
    }
    return 0;
}
```

(c) Program to be optimized.

Figure 29: Example of retrieval-based prompting. To optimized the program in Figure 29c, our dynamic prompting method retrieves the closest source program from the training set (Figure 29a), where the similarity is measured using CodeBertScore [Zhou et al., 2023b]. The slow program and the corresponding fast program (Figure 29b) from the training set are used as prompts.

network makes perfect predictions ($\hat{y}_c = 1$). As noted by Jacobs et al. [1991] (Section 1), this might cause the network to specialize slower, as the gradient will be small for experts that are

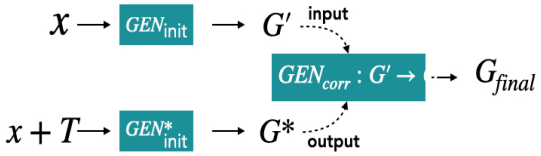



Figure 30: Training data generation to train $\text{GEN}_{\text{corr}}$.

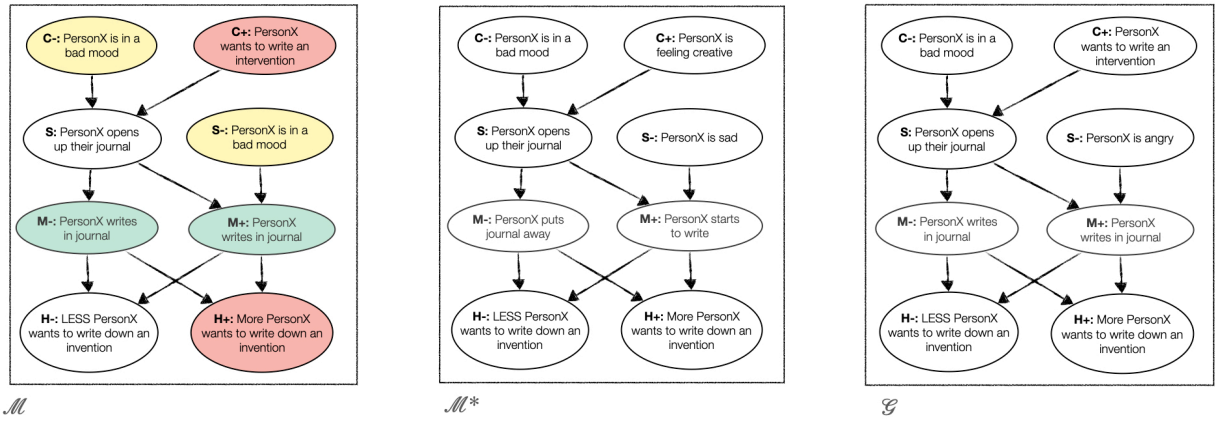



Figure 31: The graphs generated by $\text{GEN}_{\text{init}}$ (left), $\text{GEN}^*_{\text{init}}$ (middle), and $\text{GEN}_{\text{corr}}$ (right).The input graph has repetitions for nodes $\{C-, S-\}$, $\{C+, H+\}$, and $\{M-, M+\}$. The corrected graph replaces the repetitions with meaningful labels.

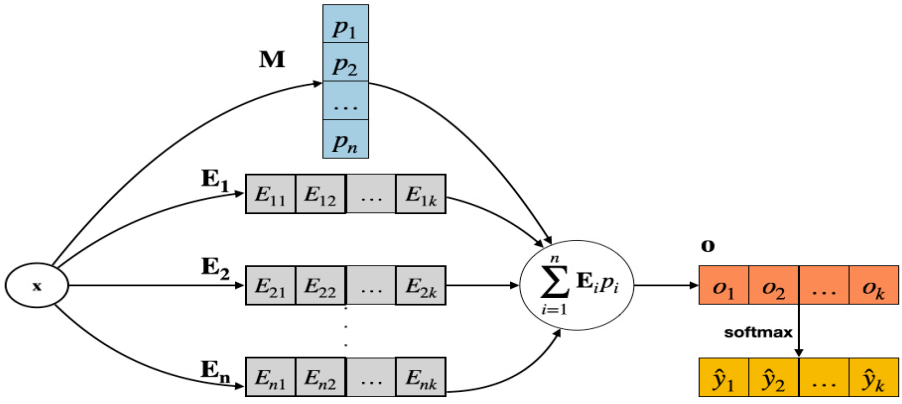



Figure 32: MOE gradient analysis setup: we consider a simple setting where the weighted output of the experts (using the expert weights $p$) is directly fed to a softmax and is used for generating class probabilities $\hat{y}$.

helping in making the correct prediction. They suggest a different loss function that promotes faster specialization by redefining the error function in terms of a mixture distribution, with the mixture weights supplied by the $p_i$ terms. Analyzing the effect of loss function for applications where the MOE is used to pool representations remains an interesting future work.

| Hyperparameter | Value |
|---|---|
| Pre-trained model | RoBERTa-base |
| Learning rate | 2e-5 |
| Gradient accumulation batches | 2 |
| Num epochs | 30 |
| Optimizer | AdamW |
| Dropout | 0.1 |
| Learning rate scheduling | linear |
| Warmup | 3 epochs |
| Batch size | 16 |
| Weight decay | 0.01 |
| Gradient clipping | 1.0 |

Table 23: General hyperparameters used by all the models.

| Hyperparameter | Value |
|---|---|
| # Layers | 2 |
| Layer dropout | 0.1 |
| Number of attention heads | 1 |
| Attention dimension | 256 |

Table 24: Hyperparameters specific to GCN.

## E.3 Hyperparameters

**Training details** All of our experiments were done on a single Nvidia GeForceRTX 2080 Ti. We base our implementation on PyTorch Paszke et al. [2017] and also use PyTorch Lightning Falcon [2019] and Huggingface Wolf et al. [2019]. The gates and the experts in our MOE model were a single layer MLP. For the experts, we set the input size set to be the same as output size. Table 23 shows the parameters shared by all the methods, and 24 shows the hyperparameters applicable to GCN encoder.

## E.4 Schema of an influence graph

Figure 33 shows the skeleton of an influence graph.

## E.5 Runtime Analysis

Finally, we discuss the cost-performance tradeoffs for various encoding mechanisms (Table 25). As Table 25 shows, both GCN and MOE take about 7% more number of parameters than the STR encoding scheme and have about 2x the runtime. Further, as we use one expert per node, the

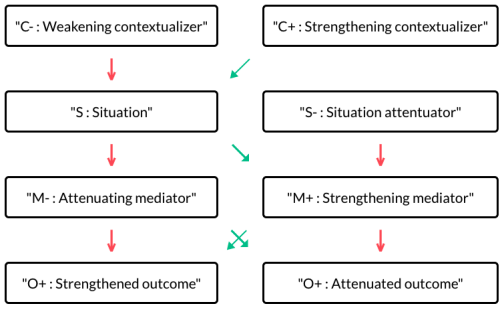



Figure 33: Schema of an inference graph.

number of parameters scales linearly with the number of nodes. While this is not prohibitive in our setting (each graph has a small number of nodes), our analysis shows that the behavior of the nodes that have similar semantics is correlated, indicating that the experts for those nodes can share parameters. Alternatively, MOE with more than two layers Jordan and Xu [1995] can also help in scaling the number of parameters only logarithmically with the number of nodes.

| Method | STR | GCN | MOE |
|---|---|---|---|
| #Params | 124M | 131M | 133M |
| Runtime | 0.17 | 0.47 | 0.40 |

Table 25: Number of parameters in the different encoding methods. Runtime reports the number of seconds to process one training example.

## E.6 Error Analysis Examples

We show three examples with different types of errors. These examples are taken from Dev set, and these are for the cases where CURIOUS introduced a wrong answer, while baseline answered this correctly without the graph.

- Figure 34 shows a failure case when a good graph is unused. Example from $\delta$-ATOMIC dev set.
- Figure 35 shows a failure case when an off topic graph is produced due to confusion in the sense of water fountain. Example from $\delta$-SNLI dev set.
- Figure 36 shows a failure case when the mediator is wrong. Example from $\delta$-SOCIAL dev set.

## E.7 Significance Tests

We perform two statistical tests for verifying our results: i) The micro-sign test (s-test) Yang and Liu [1999], and ii) McNermar's test Dror et al. [2018].

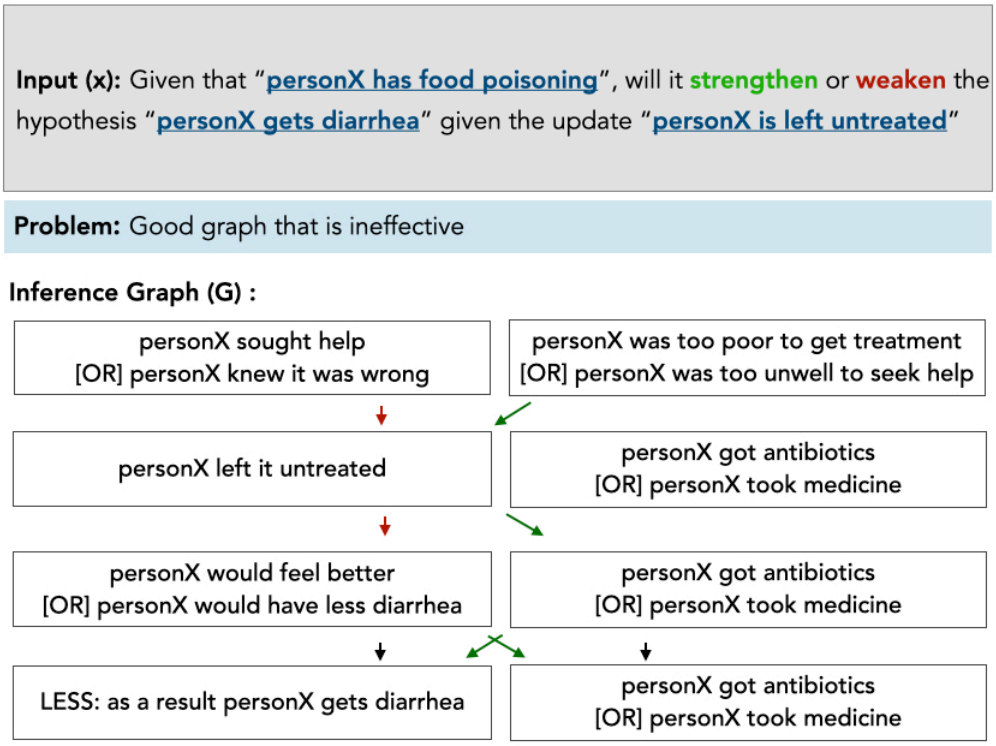



Figure 34: Example of a failure case: A good graph is unused. Example from $\delta$-ATOMIC dev set.

| Dataset | s-test | McNemar's test |
|---|---|---|
| $\delta$-ATOMIC | 5.07e-05 | 1.1e-04 |
| $\delta$-SNLI | 2.65e-05 | 6.5e-05 |
| $\delta$-SOCIAL | 1.4e-04 | 3.2e-04 |

Table 26: p-values for the three datasets and two different statistical tests while comparing the results with and without graphs (Table 6.2). As the p-values show, the results in Table 6.2 are highly significant

| Dataset | s-test | McNemar's test |
|---|---|---|
| $\delta$-ATOMIC | 0.001 | 0.003676 |
| $\delta$-SNLI | 0.01 | 0.026556 |
| $\delta$-SOCIAL | 0.06 | 0.146536 |

Table 27: p-values for the three datasets and two different statistical tests while comparing the results with noisy vs. cleaned graphs (Table 6.3).

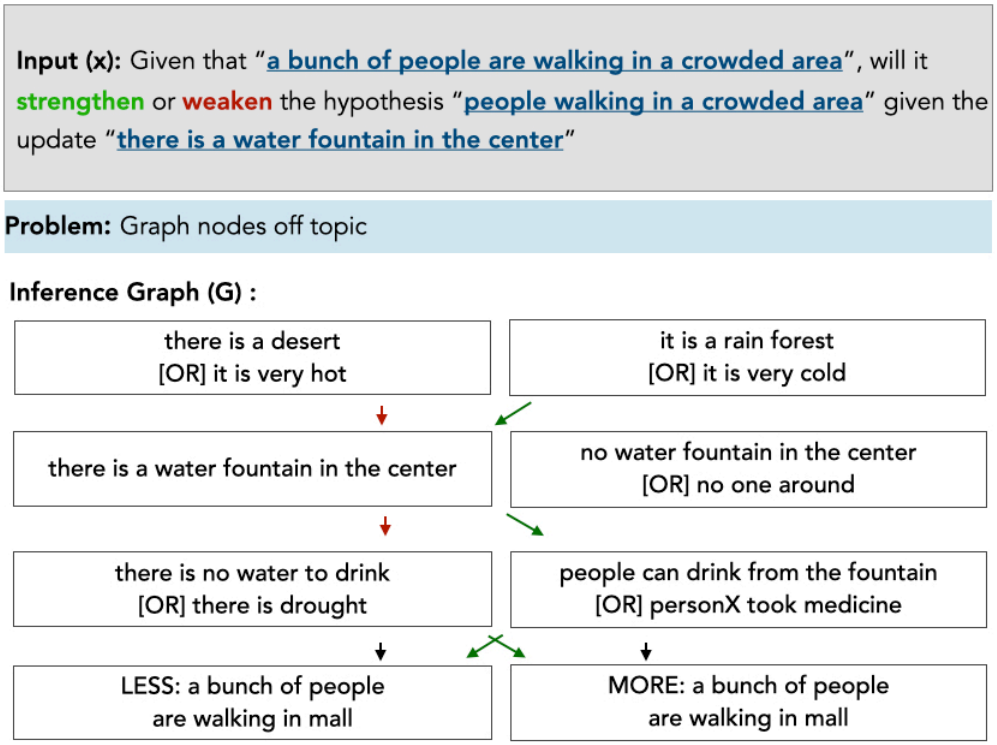



Figure 35: Example of a failure case: The generated graph is off topic (wrong sense of water fountain is used). Example from $\delta$-SNLI dev set.

| | $\delta$-ATOMIC | $\delta$-SNLI | $\delta$-SOCIAL |
|---|---|---|---|
| STR | 0.13 | 1.8e-06 | 8.7e-06 |
| GCN | 0.006 | 1.31e-05 | 0.03 |

Table 28: p-values for the s-test for Table 6.5.

| | $\delta$-ATOMIC | $\delta$-SNLI | $\delta$-SOCIAL |
|---|---|---|---|
| STR | 0.28 | 4e-06 | 2e-05 |
| GCN | 0.015127 | 3.2e-05 | 0.06 |

Table 29: p-values for the McNemar's for Table 6.5.

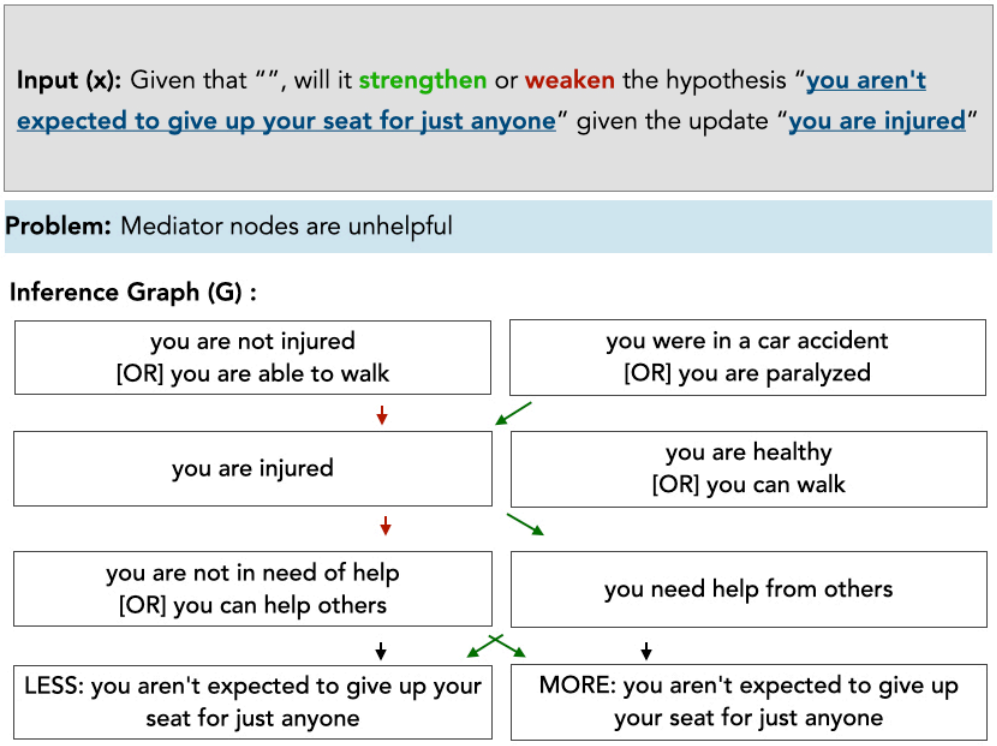



Figure 36: Example of a failure case: The mediator nodes (second last level in the graph) are unhelpful. Example from $\delta$-SOCIAL dev set.

## E.8 Description of GCN encoder

We now describe our adaptation of the method by Lv et al. [2020] to pool $\mathbf{h_V}$ into $\mathbf{h_G}$ using GCN. Figure 37 captures the overall design.

**Refining node representations** The representation for each node $v \in \mathbf{V}$ is first initialized using:

$$\mathbf{h_v^0} = \mathbf{W}^0\mathbf{h_v}$$

Where $\mathbf{h_v} \in \mathbb{R}^d$ is the node representation returned by the LMs, and $\mathbf{W}^0 \in \mathbb{R}^{d\times k}$. This initial representation is then refined by running L-layers of a GCN Kipf and Welling [2017], where each layer $l+1$ is updated by using representations from the $l^{th}$ layer as follows:

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\frac{1}{|\mathbf{A}(v)|}\sum_{w\in\mathbf{A}(v)}\mathbf{W}^l\mathbf{h}_w^l + \mathbf{W}^l\mathbf{h}_v^l\right)$$

$$\mathbf{H}^L = [\mathbf{h}_0^L; \mathbf{h}_1^L; \ldots; \mathbf{h}_{|\mathbf{V}|-1}^L] \tag{13}$$

$$\tag{14}$$

Where $\sigma$ is a non-linear activation function, $\mathbf{W}^l \in \mathbb{R}^{k\times k}$ is the GCN weight matrix for the $l^{th}$ layer, $\mathbf{A}(v)$ is the list of neighbors of a vertex $v$, and $\mathbf{H}^L \in \mathbb{R}^{|\mathbf{V}|\times k}$ is a matrix of the $L^{th}$ layer representations the $|\mathbf{V}|$ nodes such that $\mathbf{H}_i^L = \mathbf{h}_i^L$.

**Learning graph representation** We use multi-headed attention Vaswani et al. [2017] to combine the query representation $\mathbf{h_Q}$ and the nodes representations $\mathbf{H}^L$ to learn a graph representation $\mathbf{h_G}$. The multiheaded attention operation is defined as follows:

$$\begin{aligned}
\mathbf{a}_i &= \text{softmax}\left(\frac{(\mathbf{W_i^q h_Q})(\mathbf{W_i^k H}^L)^T}{\sqrt{d}}\right)\\
\text{head}_i &= \mathbf{a}_i(\mathbf{W_i^v H}^L)\\
\mathbf{h_G} &= Concat(head_1, \ldots, head_h)\mathbf{W}^O\\
&= \text{MultiHead}(\mathbf{h}_Q, \mathbf{H}^L)
\end{aligned} \tag{15}$$

Where $h$ is the number of attention heads, $\mathbf{W}_i^q, \mathbf{W}_i^k, \mathbf{W}_i^v \in \mathbb{R}^{k\times d}$ and $\mathbf{W}^O \in \mathbb{R}^{hd\times d}$.

Finally, the graph representation generated by the the MultiHead attention $\mathbf{h_G} \in \mathbb{R}^n$ is concatenated with with the question representation $\mathbf{h_Q}$ to get the prediction:

$$\hat{y} = \text{softmax}([\mathbf{h_G}, \mathbf{h_Q}]\mathbf{W}_{out})$$

where $\mathbf{W}_{out} \in \mathbb{R}^{d\times 2}$ is a single linear layer MLP.

## E.9 All results

Our experiments span two types of graphs ($\mathbf{G}'$, $G$), three datasets ($\delta$-SNLI, $\delta$-SOCIAL, $\delta$-ATOMIC), and three graph encoding schemes (STR, GCN, MOE). Table 30 above shows the results on all 18 combinations of {graph types} $\times$ {datasets} $\times$ {graph encoding schemes}

| Dataset | Encoder | Graph Type | Accuracy |
|---|---|---|---|
| | | n/a | |
| | STR | $\mathbf{G}'$ | 78.78 |
| | STR | $G$ | 79.48 |
| $\delta$-ATOMIC | GCN | $\mathbf{G}'$ | 78.25 |
| | GCN | $G$ | 78.85 |
| | MOE | $\mathbf{G}'$ | 78.83 |
| | MOE | $G$ | **80.15** |
| | | n/a | |
| | STR | $\mathbf{G}'$ | 82.16 |
| | STR | $G$ | 83.11 |
| $\delta$-SNLI | GCN | $\mathbf{G}'$ | 82.63 |
| | GCN | $G$ | 83.09 |
| | MOE | $\mathbf{G}'$ | 83.83 |
| | MOE | $G$ | **85.59** |
| | | n/a | 87.6 |
| | STR | $\mathbf{G}'$ | 86.75 |
| | STR | $G$ | 87.24 |
| $\delta$-SOCIAL | GCN | $\mathbf{G}'$ | 87.92 |
| | GCN | $G$ | 88.12 |
| | MOE | $\mathbf{G}'$ | 88.45 |
| | MOE | $G$ | **88.62** |

Table 30: Results for different combinations of graph encoder, graph type.

## E.10 Graph-augmented defeasible reasoning algorithm

In Algorithm 3, we outline our graph-augmented defeasible learning process.

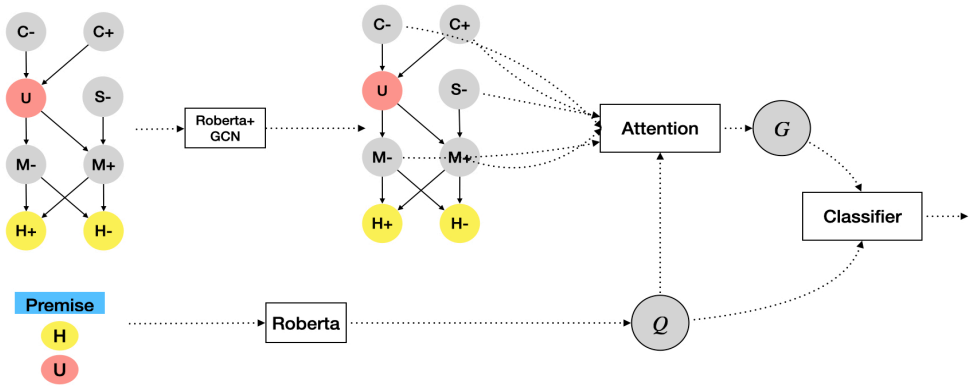



Figure 37: Overview of the GCN encoder.

**Algorithm 3** Graph-augmented defeasible reasoning using MOE

**Require:** A language model LMs, defeasible query with graph $(\mathbf{x}, \mathbf{G})$.
**Require:** Result for the query.

▷ Encode query
$\mathbf{h_Q} \leftarrow \mathcal{L}(\mathbf{x})$
▷ Encode nodes of $\mathbf{G}$
$\mathbf{h_V} \leftarrow \mathcal{L}(\mathbf{v} \in \mathbf{G})$
▷ `MOE1`: Combine nodes
$\mathbf{h_G} \leftarrow$ Equation 6.3
▷ `MOE2`: Combine $Q$, $G$
$\mathbf{h_y} \leftarrow$ Equation 6.4 **return** `softmax(MLP(`$\mathbf{h_y}$`))`

# F Chapter: Politeness Transfer: A Tag and Generate Approach 5

# G Chapter 7: Language Models of Code are Few-Shot Commonsense Learners

## G.1 Few-shot models size estimates

As OpenAI has not released any details of the size of their few-shot models, we estimate the relative strengths and weaknesses on code and text generation by calculating the average loss per token. To calculate the avg. loss of each of these models on code, we use the implementation provided by Xu et al. [2022a].[5] The perplexity on text corpus was evaluated on 30 random wikipedia pages from Wikiplots[6] following a similar procedure The structure and text generation capabilities of the models are apparent from the results in Table 33; DAVINCI outperforms CODEX on text generation but is worse on code-generation and vice-versa. CURIE underperforms both DAVINCI and CODEX significantly. Importantly, these results show that CODEX and DAVINCI are of comparable capacities, making their comparison fair.

## G.2 Dynamic prompt Creation

As an alternative to creating prompts, there is now a growing interest in customizing the in-context examples each example $\mathcal{T}_{test}$. Popular techniques typically train a retriever, which is used to fetch the examples in the training set that are closest to $\mathcal{T}_{test}$ [Liu et al., 2022a, Rubin et al., 2022, Poesia et al., 2022].

Specifically Poesia et al. [2022] train a retriever with a *target-similarity tuning* (TST) objective over a corpus of $\mathcal{D}$ of $(x, y)$ examples. TST learns an embedding function $f$ such that for a pair of examples $(x_i, y_i)$ and $(x_j, y_j)$, if $y_i \sim y_j \implies f(x_i) \sim f(x_j)$. For a new $x$, $f(x)$ is used to

[5] https://github.com/VHellendoorn/Code-LMs#evaluation
[6] https://github.com/markriedl/WikiPlots

| Non-polite Input | DRG | Our Model |
|---|---|---|
| jon - - please use this resignation letter in lieu of the one sent on friday . | - i think this would be a good idea if you could not be a statement that harry 's signed in one of the schedule . | jon - sorry - please use this resignation letter in lieu of the one event sent on |
| if you have a few minutes today, give me a call | i'll call today to discuss this. | if you have a few minutes today, please give me a call at |
| anyway you can let me know. | anyway, i'm sure i'm sure. | anyway please let me know as soon as possible |
| yes, go ahead and remove it. | yes, please go to the link below and delete it. | yes, we can go ahead and remove it. |
| can you explain a bit more about how those two coexist ? also ..... | i can explain how the two more than <unk> i can help with mike ? | can you explain a bit more about how those two coexist ? also thanks |
| go ahead and sign it - i did . | go away so we can get it approved . | we could go ahead and sign it - i did look at |

Table 31: Additional Qualitative Examples of outputs from our Model and DRG for the Politeness Transfer Task

retrieve the closest examples from $\mathcal{D}$.

We follow Poesia et al. [2022], and train a knowledge-similarity tuner (KST). We use mpnet-base[7] with SentenceTransformers [Reimers and Gurevych, 2019] to fine-tune a retrieval function $f$ by minimizing the following loss:

$$L_\theta = (\cos(f_\theta(\mathcal{T}_i), f_\theta(\mathcal{T}_j)) - \texttt{sim}(\mathcal{G}_i, \mathcal{G}_j))^2 \tag{16}$$

where $f_\theta$ is parameterized using a transformer.

Results on using KST with PROSCRIPT (Table 34) and EXPLAGRAPHS (Table 35). While KST is highly effective for edge-prediction 7.6, the results are mixed for EXPLAGRAPHS and PROSCRIPT. For PROSCRIPT, KST yields marginal gains. However, for EXPLAGRAPHS, a number of training examples have overlapping theme (Table 36), and thus creating a prompt dynamically reduces the effective information in the prompt.

## G.3 Human Evaluation

Out of the four tasks used in this work, PROSCRIPT edge prediction and PROPARA have only one possible correct value. Thus, following prior work, we report the automated, standard metrics for these tasks. For EXPLAGRAPHS, we use model-based metrics proposed by Saha et al. [2021], which were found to have a high correlation with human judgments. For PROSCRIPT

[7] https://huggingface.co/microsoft/mpnet-base

| Task | | Non-polite Input | DRG | Our Model |
|---|---|---|---|---|
| Fem Male | → | my husband ordered the brisket . | my wife had the best steak . | my wife ordered the brisket . |
| Fem Male | → | i ' m a fair person . | i ' m a good job of the <unk> . | i ' m a big guy . |
| Male Fem | → | my girlfriend and i recently stayed at this sheraton . | i recently went with the club . | my husband and i recently stayed at this office . |
| Male Fem | → | however , once inside the place was empty . | however , when the restaurant was happy hour for dinner . | however , once inside the place was super cute . |
| Pos Neg | → | good drinks , and good company . | horrible company . | terrible drinks , terrible company. |
| Pos Neg | → | i will be going back and enjoying this great place ! | i will be going back and enjoying this great ! | i will not be going back and enjoying this garbage ! |
| Neg Pos | → | this is the reason i will never go back . | this is the reason i will never go back . | so happy i will definitely be back . |
| Neg Pos | → | salsa is not hot or good . | salsa is not hot or good . | salsa is always hot and fresh . |
| Dem Rep | → | i am confident of trumps slaughter . | i am mia love | i am confident of trumps administration . |
| Dem Rep | → | we will resist trump | we will impeach obama | we will be praying for trump |
| Rep Dem | → | video : black patriots demand impeachment of obama | video : black police show choose | video : black patriots demand to endorse obama |
| Rep Dem | → | mr. trump is good ... but mr. marco rubio is great ! ! | thank you mr. good ... but mr. kaine is great senator ! ! | mr. schumer is good ... but mr. pallone is great ! ! |
| Fact Rom | → | a woman is sitting near a flower bed overlooking a tunnel . | a woman is sitting near a flower overlooking a tunnel, determined to | a woman is sitting near a brick rope , excited to meet her boyfriend . |
| Fact Rom | → | two dogs play with a tennis ball in the snow . | two dogs play with a tennis ball in the snow . | two dogs play with a tennis ball in the snow celebrating their friendship . |
| Fact Hum | → | three kids play on a wall with a green ball . | three kids on a bar on a field of a date . | three kids play on a wall with a green ball fighting for supremacy . |
| Fact Hum | → | a black dog plays around in water . | a black dog plays in the water . | a black dog plays around in water looking for fish . |



Table 32: Additional Qualitative Examples of our Model and DRG for other Transfer Tasks

| Model | CODE | TEXT |
|---|---|---|
| CODEX | **0.46** | 2.71 |
| DAVINCI | 0.63 | **2.25** |
| CURIE | 1.17 | 3.32 |

Table 33: Average loss per token of the three few-shot models used in this work. TEXT refers to the average loss over 30 Wikipedia pages, and CODE is the loss over Python scripts in the evaluation split of Polycoder.

| | ISO | GED | Avg(d) | Avg($\|\boldsymbol{V}\|$) | Avg($\|\boldsymbol{E}\|$) | BLEU | ROUGE-L | BLEURT | |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{G}$ | 1.0 | 0.0 | 1.84 | 7.41 | 6.8 | - | - | - | - |
| GPT-2 + 002 (15) | 0.53 | 2.1 | 1.79 | **7.44** | **6.7** | 25.24 | 38.28 | -0.26 | |
| GPT-2 + 002 (15) + KST | 0.52 | 1.99 | **1.8** | 7.45 | **6.7** | **25.4** | **38.4** | **-0.25** | |

Table 34: KST on PROSCRIPT generation: Dynamically creating a prompt leads to marginal improvements.

graph generation, we conducted an exhaustive automated evaluation that separately scores the correctness of the nodes and the correctness of the edges.

However, automated metrics are limited in their ability to evaluate model-generated output. Thus, to further investigate the quality of outputs, we conduct a human evaluation to compare the outputs generated by GPT-2 and DAVINCI. We sampled 20 examples, and three of the authors performed the evaluation. Annotators were shown two graphs (generated by GPT-2 and DAVINCI) and were asked to select one they thought was better regarding relevance and correctness. The selection for each criterion was made independently: the same graph could The annotations were done separately: the same graph could have more relevant nodes (higher relevance) but may not be correct. The identity of the model that generated each graph (GPT-2 or DAVINCI) was shuffled and unknown to the evaluators.

The results in Table 37 indicate that human evaluation is closely correlated with the automated metrics: for EXPLAGRAPHS, annotators found the graphs generated by GPT-2 to be more relevant and correct. We find that DAVINCI often fails to recover semantic relations between nodes in the argument graphs. For example, consider a belief (B) *urbanization harms natural habitats for the animals in the world*. We want to generate a graph that can **counter** this belief with the argument

| | StCA (↑) | SeCA (↑) | G-BS (↑) | GED (↓) | EA (↑) |
|---|---|---|---|---|---|
| GPT-2 + 002 | **45.2** | **23.74** | **34.68** | **68.76** | **23.58** |
| GPT-2 + 002 + KST | 37.47 | 18.46 | 29.41 | 73.76 | 19.15 |

Table 35: KST on EXPLAGRAPHS: We find that EXPLAGRAPHS contains multiple examples that are similar to each other in the training set. Thus, dynamically creating a prompt by selecting examples that are closest to the input actually hurts performance.

| |
|---|
| *belief*: all religions need to be respected, and able to practice. *argument*: religion is behind many wars. |
| *belief*: every religion needs to be respected and allowed to be practiced. *argument*: religion is behind most wars. |
| *belief*: school prayer should not be allowed. *argument*: many people would prefer to keep religion out of their lives. |
| *belief*: people should follow whichever religion they choose. *argument*: this country has freedom of religion. |
| *belief*: people are free to practice the religion they choose*argument*: society's right to be free to practice religion should not be limited. |
| *belief*: the church of scientology should be allowed, because everyone has a right to follow their religion. *argument*: the church of scientology doesn't have a religious doctrine. |
| *belief*: we should avoid discussing religion in schools.*argument*: some schools are religious in nature, and have regular discussions on the topic. |
| *belief*: freedom of religion is paramount. *argument*: not all religions are worth it. |
| *belief*: people don't follow the same religion. *argument*: the world has many different religions. |
| *belief*: people should follow whatever religion they desire. *argument*: people have the right to adhere to the religion of their choice. |
| *belief*: people should follow whichever religion they choose. *argument*: some religions are better than others. |
| *belief*: people should be able to practice whatever religion they choose. *argument*: some religions are not okay to pursue. |
| *belief*: students have a right to express themselves any way possible, including faith. *argument*: religion is a personal choice. |
| *belief*: people should be able to do missionary work if they desire. *argument*: people should have right to missionary work. |
| *belief*: students are free to express faith. *argument*: one should go to church to express their religious beliefs. |

Table 36: The closest examples in the training set corresponding to the test input: *belief*: religion causes many fights. and *argument*: *There would be less fights without religious conflicts.*. As the table shows, the examples are overlapping which reduces the diversity in the prompt, effectively reducing the number of examples in a prompt creating using nearest neighbors (Section 7.4.

| | Dataset | GPT-2 | DAVINCI | No preference |
|---|---|---|---|---|
| Relevance | EXPLAGRAPHS | 28.3% | 16.7% | 46.7% |
| | PROSCRIPT (script generation) | 26.7% | 18.3% | 55% |
| Correctness | EXPLAGRAPHS | 38.3% | 18.3% | 31.7% |
| | PROSCRIPT (script generation) | 26.7% | 23.3% | 50% |

Table 37: Human evaluation of graphs generated by GPT-2 and DAVINCI. The evaluators were shown graphs generated by GPT-2 and DAVINCI, and were asked to select one that is more relevant to the input and correct. In case of no preference, the evaluators could pick the No preference. The table shows the % of times graphs from each model were preferred.

(A) *urbanization causes increase in jobs.*

For the same prompt, GPT-2 generated *(urbanization; causes; increase in jobs); (increase in jobs; has context; good); (good; not capable of; harms)* whereas DAVINCI generated *(jobs; not harms; natural habitats)* $\rightarrow$ *(natural habitats; not part of; animals).* Note that DAVINCI successfully recovered relevant events ("natural habitat" "animals") but arranged them in incorrect relations. For PROSCRIPT, the human evaluation shows that GPT-2 and DAVINCI have complementary strengths, while GPT-2 generally produces more relevant and correct outputs.

## G.4 Dataset statistics

Dataset statistics are shown in Table 38. The test split for EXPLAGRAPHS is not available, so we evaluate on the validation split. For PROSCRIPT, we obtained the test splits from the authors.

| **Corpus** | **#Train** | **#Val** | **#Test** |
|---|---|---|---|
| PROSCRIPT | 3252 | 1085 | 2077 |
| PROPARA | 387 | 43 | 54 |
| EXPLAGRAPHS | 2368 | 398 | - |

Table 38: Corpus Statistics for the tasks used in this work.

## G.5 Sample outputs

Sample outputs from GPT-2 for all the tasks are located at https://github.com/madaan/CoCoGen/tree/main/outputs. Representative examples from each task are presented in Figure 38. Surprisingly, GPT-2 (CODEX with a Python prompt) generates syntactically valid Python graphs that are similar to the task graphs/tables in nearly 100% of the cases.

## G.6 Prompts

The prompts for each tasks are present at this anonymous URL:

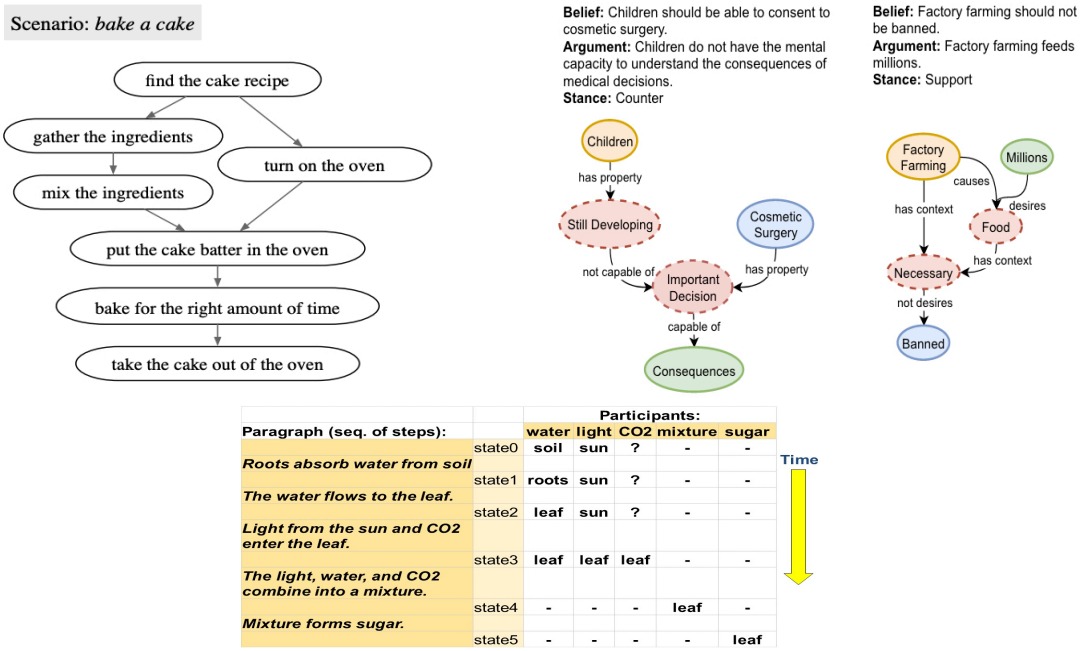



Figure 38: Example graphs for each of the tasks used for GPT-2: PROSCRIPT (top-left), EXPLAGRAPHS (top-right), and PROPARA (bottom).

1. PROSCRIPT script-generation: https://github.com/madaan/CoCoGen/tree/main/data/proscript_script_generation/prompt.txt
2. PROSCRIPT edge-prediction: https://github.com/madaan/CoCoGen/tree/main/data/proscript_edge_prediction/prompt.txt
3. PROPARA: https://github.com/madaan/CoCoGen/tree/main/data/explagraphs/prompt.txt
4. EXPLAGRAPHS: https://github.com/madaan/CoCoGen/tree/main/data/explagraphs/prompt.txt

These prompts are also present in the attached supplementary material, and can be found in the `data` folder under respective task sub-directories.

## G.7 Designing Python class for a structured task

Figure 40 shows three different designs for Explagraphs. For PROSCRIPT, the various formats include representing proscript as a Networkx[8] class (41), DOT-like class 42, and as a Tree (43).

[8] https://networkx.org/

| Model | Format | StCA (↑) | SeCA (↑) | G-BS (↑) | GED (↓) | EA (↑) |
|---|---|---|---|---|---|---|
| CODEX-002 | Literal | **45.2** | **23.74** | **34.68** | **68.76** | **23.58** |
| CODEX-002 | Tree | 39.24 | 15.95 | 30.49 | 73.85 | 18.24 |
| CODEX-002 | Relation | 42.82 | 23.68 | 33.38 | 70.23 | 21.16 |

Table 39: Performance of CODEX on the three different formats present in Figure 40 for EXPLAGRAPHS.

| Model | Format | $F_1$ |
|---|---|---|
| CODEX-001 | Literal | 15.9 |
| CODEX-001 | Tree | 29.7 |
| CODEX-002 | Literal (Figure 42) | 52.0 |
| CODEX-002 | Tree (Figure 43) | 56.5 |

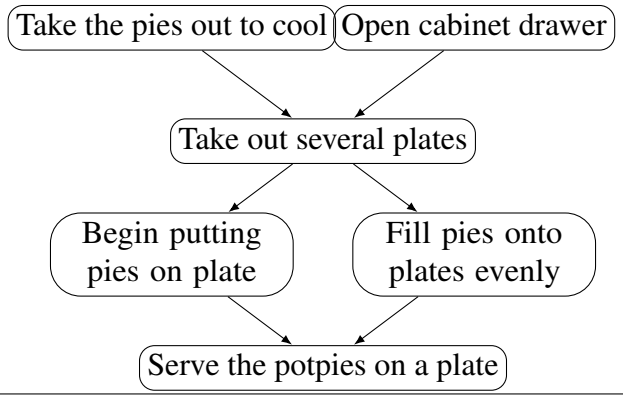
Table 40: Performance of CODEX-001 and CODEX-002 on the the different formats present in Figure 43 and 42 for PROSCRIPT edge prediction. We find that the literal format that combines structure with literally Figure output performs the best for CODEX-002.

| Model | Format | ISO | GED | Avg(d) | Avg($\|\boldsymbol{V}\|$) | Avg($\|\boldsymbol{E}\|$) | BLEU | ROUGE-L | BLEURT |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{G}$ | - | 1.0 | 0.0 | 1.84 | 7.41 | 6.8 | - | - | - |
| CODEX-001 | Literal (Figure 42) | **0.55** | **1.8** | 1.74 | 7.45 | 6.5 | 22.9 | 36.2 | -0.36 |
| CODEX-001 | Tree (Figure 43) | 0.35 | 3 | **1.79** | 7.45 | 6.65 | 17.8 | 30.7 | -0.45 |
| CODEX-001 | NetworkX (Figure 41) | 0.51 | 1.81 | 1.69 | 7.49 | 6.32 | 23.7 | 35.9 | -0.37 |
| CODEX-002 | Literal (Figure 42) | 0.53 | 2.1 | **1.79** | **7.44** | **6.7** | **25.24** | **38.28** | **-0.26** |
| CODEX-002 | Tree (Figure 43) | 0.35 | 2.46 | 1.61 | 7.46 | 5.74 | 18.96 | 32.92 | -0.38 |
| CODEX-002 | NetworkX (Figure 41) | 0.5 | 2.46 | **1.79** | **7.38** | 6.61 | 23.88 | 36.89 | -0.33 |

Table 41: CODEX results on PROSCRIPT generation for various Python source formats.

| | ISO | GED | Avg(d) | Avg($\|\boldsymbol{V}\|$) | Avg($\|\boldsymbol{E}\|$) | BLEU | ROUGE-L | BLEURT |
|---|---|---|---|---|---|---|---|---|
| $\mathcal{G}$ | 1.0 | 0.0 | 0.0 | 1.84 | 7.41 | 6.8 | - | - - |
| GPT-2 + 001 (15) | 0.55 | 1.8 | 1.74 | 7.45 | 6.5 | 22.9 | 36.2 | -0.36 |
| GPT-2 + 002 (15) | 0.53 | 2.1 | 1.79 | **7.44** | **6.7** | **25.24** | **38.28** | **-0.26** |

Table 42: CODEX-001 vs 002 on PROSCRIPT script generation




```
class Tree:

  goal = "serve the potpies on a plate"

  def __init__(self):
    # nodes
    begin = Node()
    take_pies_out_to_cool = Node()
    take_out_several_plates = Node()
    open_cabinet_drawer = Node()
    fill_pies_onto_plates_evenly = Node()
    begin_putting_pies_on_plates = Node()
    serve_potpies_on_plate = Node()

    # edges
    begin.children = [take_pies_out_to_cool, open_cabinet_drawer]
    take_pies_out_to_cool.children = [take_out_several_plates]
    open_cabinet_drawer.children = [take_out_several_plates]
    take_out_several_plates.children = [begin_putting_pies_on_plates,
        fill_pies_onto_plates_evenly]
    begin_putting_pies_on_plates.children = [serve_potpies_on_plate]
    fill_pies_onto_plates_evenly.children = [serve_potpies_on_plate]
    serve_potpies_on_plate.children = [end]
```

Figure 39: A PROSCRIPT plan (top) and the corresponding Python code (bottom).

## G.8 Impact of Model size

The CODEX model released by OpenAI is available in two versions[9]: `code-davinci-001` and `code-davinci-002`. While the exact sizes of the models are unknown because of their proprietary nature, OpenAI API states that `code-davinci-002` is the *Most capable Codex model* Table 42 compares GPT-2 +`code-davinci-001` with GPT-2 +`code-davinci-002`. Note that both `code-davinci-001` and `code-davinci-002` can fit 4000 tokens, so the number of in-context examples was identical for the two settings. The results show that for identical prompts, GPT-2 +`code-davinci-002` vastly outperforms GPT-2 +`code-davinci-001`,

[9] as of June 2022

```
class Relation:

    def __init__(self):
        belief = "Cannabis should be legal."
        argument = "It's not a bad thing to make marijuana more available."
        stance = "support"

        # create a DAG to support belief using argument
        begin = ["cannabis"]
        add_edge("cannabis", "synonym of", "marijuana")
        add_edge("legal", "causes", "more available")
        add_edge("marijuana", "capable of", "good thing")
        add_edge("good thing", "desires", "legal")

class Tree:
    def __init__(self):
        self.belief = "Cannabis should be legal."
        self.argument = "It's not a bad thing to make marijuana more available."
        self.stance = "support"

        # tree for support in support of belief
        root_nodes = cannabis
        cannabis = Node()
        cannabis.add_edge("synonym of", "marijuana")
        legal = Node()
        legal.add_edge("causes", "more available")
        marijuana = Node()
        marijuana.add_edge("capable of", "good thing")
        good_thing = Node()
        good_thing.add_edge("desires", "legal")


 class Literal:
    def __init__(self):
        self.belief = "Cannabis should be legal."
        self.argument = "It's not a bad thing to make marijuana more available."
        self.stance = "support"
        self.graph = """\
(cannabis; synonym of; marijuana)(legal; causes; more available)
(marijuana; capable of; good thing)
(good thing; desires; legal)"""
```

Figure 40: Template candidates for EXPLAGRAPHS.

```
class Plan:

    goal = "create a video game"
    num_steps = 7

    def __init__(self):
        graph = nx.DiGraph()
        # add nodes
        step0 = "decided to create a video game"
        step1 = "Learn the basics of programming"
        step2 = "Learn to use a language that is used in games"
        step3 = "Learn to use an existing game engine"
        step4 = "Program the game"
        step5 = "Test the game"
        step6 = "create a video game"
        graph.add_nodes_from([step0, step1, step2, step3, step4, step5, step6])

        # add edges
        graph.add_edge(step0, step1)
        graph.add_edge(step1, step2)
        graph.add_edge(step1, step3)
        graph.add_edge(step2, step4)
        graph.add_edge(step3, step4)
        graph.add_edge(step4, step5)
        graph.add_edge(step5, step6)
```

Figure 41: Proscript as a Networkx class.

```
class CreateAVideoGame:

   title = "create a video game"
   steps = 7

   def step0(self):
       return "decided to create a video game"
   def step1(self):
       return "Learn the basics of programming"
   def step2(self):
       return "Learn to use a language that is used in games"
   def step3(self):
       return "Learn to use an existing game engine"
   def step4(self):
       return "Program the game"
   def step5(self):
       return "Test the game"
   def step6(self):
       return "create a video game"
   def get_relations(self):
       return [
           "step0 -> step1",
           "step1 -> step2",
           "step1 -> step3",
           "step2 -> step4",
           "step3 -> step4",
           "step4 -> step5",
           "step5 -> step6",
       ]
```

Figure 42: Representing PROSCRIPT graph literally.

```
class Tree:

    goal = "serve the potpies on a plate"

    def __init__(self):
        # nodes
        begin = Node()
        take_pies_out_to_cool = Node()
        take_out_several_plates = Node()
        open_cabinet_drawer = Node()
        fill_pies_onto_plates_evenly = Node()
        begin_putting_pies_on_plates = Node()
        serve_potpies_on_plate = Node()

        # edges
        begin.children = [take_pies_out_to_cool, open_cabinet_drawer]
        take_pies_out_to_cool.children = [take_out_several_plates]
        open_cabinet_drawer.children = [take_out_several_plates]
        take_out_several_plates.children = [begin_putting_pies_on_plates,
                                            fill_pies_onto_plates_evenly]
        begin_putting_pies_on_plates.children = [serve_potpies_on_plate]
        fill_pies_onto_plates_evenly.children = [serve_potpies_on_plate]
        serve_potpies_on_plate.children = [end]
```

Figure 43: Proscript with a tree-encoding.

showing the importance of having a better underlying code generation model.

**Model size vs. sensitivity to the prompt** In Table 40 shows the performance of CODEX-001 (smaller) and CODEX-002 (larger, also see Appendix G.1) on identical prompts. Our experiments show that as model size increases, the sensitivity of the model on the prompt reduces. This indicates that for very large models, prompt design might get progressively easier.

## G.9 Variation in prompts

We run each experiment with 3 different random seeds, where the random seeds decides the order of examples in the prompt. We find minimal variance between runs using different fixed prompts between 3 runs. Further, as shown in the Tables 44,45, 46, and 47, all improvements of GPT-2 over DAVINCI are statistically significant (p-value $< 0.001$).

| | BLEU | ROUGE-L | BLEURT |
|---|---|---|---|
| DAVINCI | 23.1±2.7 | 36.5±2.7 | -0.27±0.06 |
| GPT-2 | 25.3±0.1 | 38.3±0.1 | -0.25±0.01 |

Table 44: PROSCRIPT script generation: mean and standard deviation across three different random seeds.

| | StCA (↑) | SeCA (↑) | G-BS (↑) | GED (↓) | EA (↑) |
|---|---|---|---|---|---|
| DAVINCI | 25.4 ± 2.7 | 13.7 ± 2.8 | 20 ± 2.3 | 82.5 ± 1.9 | 13.6 ± 1.8 |
| GPT-2 | **44.0** ± 1.2 | **25.1** ± 2.5 | **34.1** ± 0.7 | **69.5** ± 0.7 | **22.0** ± 1.3 |

Table 46: EXPLAGRAPHS: mean and standard deviation across three different random seeds.

| | F1 |
|---|---|
| DAVINCI | 48.9 ± 2.8 |
| GPT-2 | **56.2±2.1** |

Table 45: PROSCRIPT edge prediction: mean and standard deviation across three different random seeds.

| | F1 |
|---|---|
| DAVINCI | 56.9 ± 2.4 |
| GPT-2 | **62.8** ± 2.4 |

Table 47: PROPARA: mean and standard deviation across three different random seeds.

# H Chapter 8: Program Aided Language Models

## H.1 Alternative Prompts without Meaningful Variable Names

```
a = 23
b = 5
c = 3
d = b * c
e = a - d
print(e)
```

(a) Structured explanation with uninformative variable names (PAL - `var`)

```
# Olivia has $23
a = 23
# number of bagels bought
b = 5
# price of each bagel
c = 3
# total price of bagels
d = b * c
# money left
e = a - d
print(e)
```

(b) Structured explanation with uninformative variable names, but useful comments (PAL - `var` + `comms`)

```
money_initial = 23
bagels = 5
bagel_cost = 3
money_spent = bagels * bagel_cost
money_left = money_initial - money_spent
result = money_left
print(result)
```

(c) PAL prompts

Figure 44: **Role of text in PAL**: three different reasoning steps for the question *Olivia has $23. She bought five bagels for $3 each. How much money does she have left?* Uninformative variable names (left), Uninformative variable names with useful comments (left), and PAL. Including text description

For mathematical problems, since our standard prompts do not use much comment, we start by creating alternative prompts where the informative variable names are replaced with single-letters (Figure 44). The results in Table 48 shows a considerable performance drop: from an average of 71.8% to 59%. Note that the ablation where structured outputs are completely removed in favor of purely text explanations is precisely the COT setting, which achieves a solve rate of 63%. These results underscore the importance of text but more importantly show that combining both text and procedural statements leads to higher performance gains—either is sub-optimal.

| Setting | CoT | PaL - `var` | PaL - `var` + `comms` | PaL |
|---|---|---|---|---|
| Solve Rate | 63.1 | 59.0 | 69.0 | 71.8 |

Table 48: Role of text: including text either as informative variable names (PaL) or comments is important (PaL - `var` + `comms`). Uninformative variable names PaL - `var` cause a drastic drop in performance, indicating that just structure is not sufficient. The corresponding prompts are shown in Figure 44.

## H.2 Additional analysis on Arithmetic Reasoning

**GSM-hard with hard prompts** The GSM-HARD experiments used prompts that were sampled from the Math Reasoning training set. Will CoT be helped by using larger numbers in the prompts as well? To investigate this, we create prompts where the numbers are changed to larger numbers, matching the distribution of numbers in GSM-HARD. The results in Table 49 shows that even with a prompt that matches the numbers, there are only modest gains in performance. These results show that the gains achieved by using code-based reasoning chains may not be achieved simply by using better few-shot examples for CoT.

| | Regular Prompt | Prompt with Larger Numbers |
|---|---|---|
| CoT | 23.3 | 23.8 |

Table 49: GSM-hard results, when the prompts also had examples of larger numbers.

**Succinct code** The programs used in few-shot examples by PaL are multi-step, and show a step-by-step breakdown of the reasoning process. Is this breakdown necessary? Alternatively, can we return a single line expression (see Figure 45b) to calculate the result? Results in Table 50 ($4^{th}$ row) shows that is not the case. With single-line expressions, the performance of PaL falls to the level of direct prompting.

**Generating the answer directly** PaL first generates a reasoning chain in the form of a Python program, and passes the generated program to a runtime to obtain an answer. Is PaL better *only* because of the program-style intermediate reasoning chains, or are the improvements derived from offloading execution to the Python runtime? To investigate this, we experiment with a variant that forces the LLM to generate the answer after generating the reasoning chain (Figure 45e). This setting compels the LLM to condition on the generated code-based reasoning to generate an answer, simulating the runtime. The results in Table 50 ($5^{th}$ row) show that the solve rate drops to near DIRECT levels. This reinforces our hypothesis that while current LLMs can be excellent at specifying a high-level plan to solve a task—they are still incapable of executing them.

| Ablation | Solve Rate |
|---|---|
| DIRECT (no intermediate reasoning) | 19.7 |
| COT | 65.6 |
| PAL | 72.0 |
| Succinct Code | 47.8 |
| LLM Simulating Runtime | 23.2 |

Table 50: Solve rates for ablations

## H.3 Effect of Using Language Models of Code

In our experiments, we focused on evaluating the performance of a language model for code. We aimed to investigate whether the additional performance boost observed in our results was due to the use of models like Codex, or whether our formulation was useful even for text-based models. To this end, we conducted additional experiments using text-based language models. Our findings indicate that the PAL approach is not restricted to working solely with Codex, but can also be applied to natural language (NL) models, as long as the model is sufficiently strong. Specifically, our results showed that in the text-davinci-001 model, the use of the CoT approach resulted in better performance.

| Model | CoT | PaL |
|---|---|---|
| text-davinci-001 | 26.5 | 8.6 |
| text-davinci-002 | 46.9 | 65.8 |
| text-davinci-003 | 65.3 | 69.8 |

Table 51: Performance on Math Reasoning with different language models of text

## H.4 Experiments with ChatGPT

We compare CoT and PaL on Math Reasoning with ChatGPT (`gpt-turbo-3.5`), a dialogue-tuned language model. We follow the official guideline[10] to present few-shot examples as the `example_user` and the `example_assistant`. The results are shown in Table 52. We find that PAL achieves stronger performance of 79.6, outperforming CoT by 2.8%.

| Model | CoT | PaL |
|---|---|---|
| ChatGPT | 76.8 | 79.6 |

Table 52: Performance on Math Reasoning with ChatGPT

[10] https://github.com/openai/openai-cookbook

## H.5 Analyzing the Effect of Increasing Number of Samples on PAL

In Section 8.5.1, we show that PAL outperforms strong baselines both for a single sample and by drawing 40 samples and using majority voting. Figure 46 illustrates the trends for cases when the number of samples drawn are between 1 and 40, and the interpolation estimates demonstrate that PAL remains competitive throughout the number of samples.

```
def solution():
"""Shawn has five toys. For Christmas, he got two toys each
 ;  from his mom and dad. How many toys does he have now?"""
    toys_initial = 5
    mom_toys = 2
    dad_toys = 2
    total_received = mom_toys + dad_toys
    total_toys = toys_initial + total_received
    result = total_toys
    return result
```

(a) Original Example

```
def solution():
    return 5 + 2 + 2
```

(b) Succinct Code

```
def solution():
"""Shawn has 10312864 toys. For Christmas, he got 13267894
    toys each from his mom and dad. How many toys does he
    have now?"""
    toys_initial = 10312864
    mom_toys = 13267894
    dad_toys = 13267894
    total_received = mom_toys + dad_toys
    total_toys = toys_initial + total_received
    result = total_toys
    return result
```

(c) Hard Examples in Prompt (PAL)

```
Example(
    question="Shawn has 10312864 toys. For Christmas, he got
        13267894 toys each from his mom and dad. How many
        toys does he have now?",
    thought="Shawn started with 10312864 toys. If he got
        13267894 toys each from his mom and dad, then that is
        26535788 more toys. 10312864 + 26535788 =
        36848652.",
    answer="36848652",
),
```

(d) Hard Examples in Prompt (CoT)

```
def solution():
"""Shawn has five toys. For Christmas, he got two toys each
 ;  from his mom and dad. How many toys does he have now?"""
    toys_initial = 5
    mom_toys = 2
    dad_toys = 2
    total_received = mom_toys + dad_toys
    total_toys = toys_initial + total_received
    result = total_toys
    return result
    ans = 9
```

(e) Generating Answers Directly

Figure 45: Ablations of the original example solution for the few-shot prompting experiment.

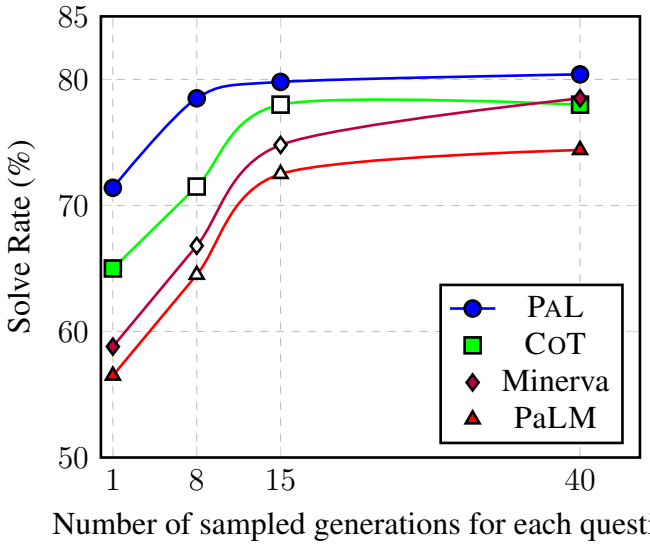



Figure 46: Comparison of solve rates between PAL and baselines as the number of samples increases from 1 to 40. Note that the solve rates for the baselines (PaLM, CoT, Minerva) are obtained through logistic interpolation of solve rates at 1 and 40

**Standard Deviations Across Multiple Order of Prompts**

For each math reasoning task, we run inference using three random orderings of the prompts. As shown in Table 53, the standard deviation between the results obtained from the three different seeds is minimal.

| | CoT | | PaL | |
|---|---|---|---|---|
| | Average | Standard Deviation | Average | Standard Deviation |
| Math Reasoning | 65.6 | 1.10 | 72.0 | 0.16 |
| SVAMP | 74.8 | 0.19 | 79.4 | 0.20 |
| ASDIV | 76.9 | 0.65 | 79.6 | 0.14 |
| GSM-HARD | 23.3 | 0.49 | 61.2 | 0.91 |
| MAWPS-SingleEq | 89.1 | 0.54 | 96.1 | 0.30 |
| MAWPS-SingleOp | 91.9 | 0.55 | 94.6 | 0.36 |
| MAWPS-AddSub | 86.0 | 0.62 | 92.5 | 0.34 |
| MAWPS-MultiArith | 95.9 | 0.51 | 99.2 | 0.48 |

Table 53: Standard deviations for three runs for the math reasoning datasets.

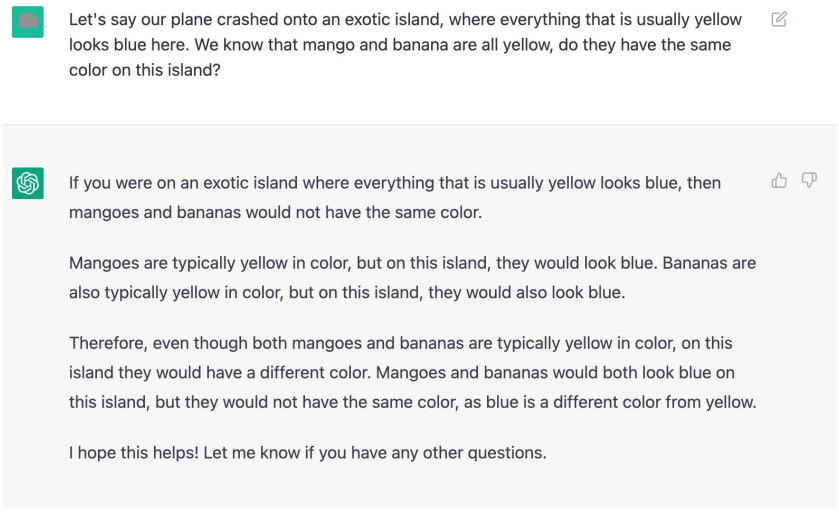
## H.6 PaL Beyond Benchmarks

We argue that symbolic reasoning is a crucial component in solving a wide range of tasks. In this section, we demonstrate examples of tasks that may not initially appear to require using programs as intermediate reasoning steps, but can be improved through the use of PaL-style reasoning. We demonstrate these examples using the ChatGPT tool.[11] In contrast to the in-context-learning methods we used in the main paper, here we instruct ChatGPT to perform program-aided reasoning through one of the user utterances.

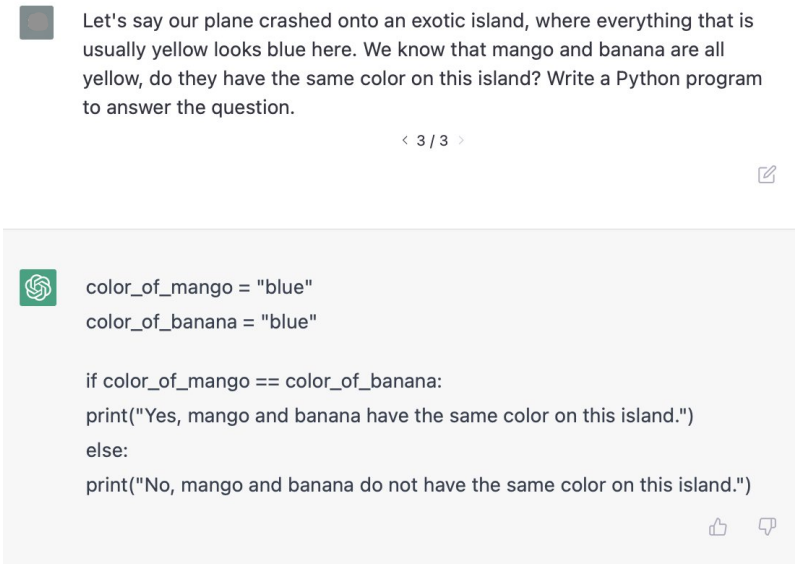
In Figure 47, in CoT-style reasoning, while the reasoning chain is correct, the final answer is wrong. In contrast, PaL-style reasoning could not only accurately extract the color of objects from the question but also produce the correct lines of code to branch to different situations that yield their corresponding correct answers.

A more intriguing example is letting an LLM count the number of letters in the word "intriguing". In Figure 48a, while the step-by-step explanation appears reasonable by splitting the letters by spaces, ChatGPT does not change the answer after this explicit reasoning and insists on the wrong answer. Explicitly instructing the model to perform step-by-step reasoning *before* answering the question still yields the wrong answer. In contrast, PaL-style reasoning only takes a few lines of code, and the execution does produce the correct answer, in this case. These examples indicate that PaL can benefit even an ostensibly powerful model like ChatGPT.

[11] chat.openai.com

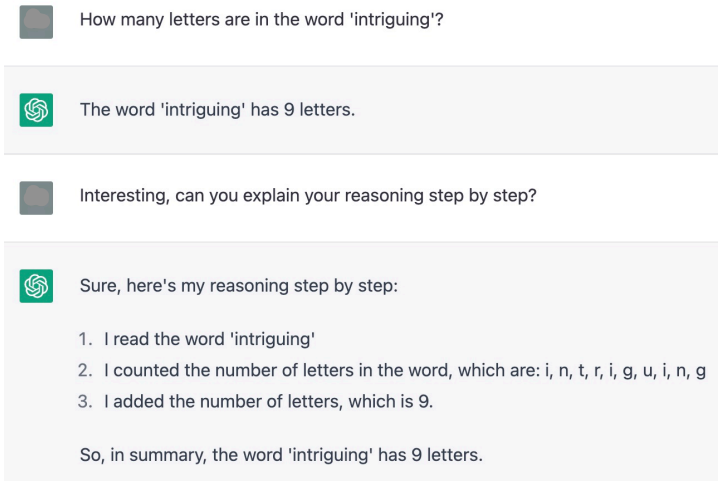



(a) In CoT style reasoning, the correct intermediate reasoning chain leads to wrong answers.

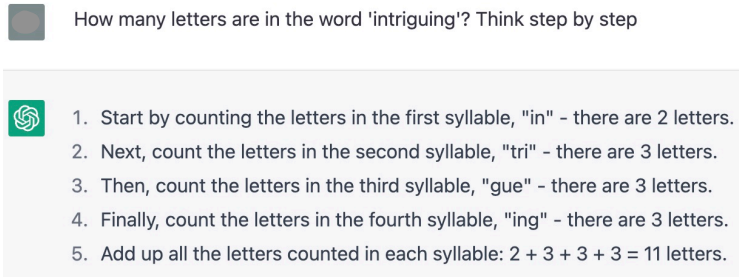



(b) In PAL, the execution of the code will produce the correct answer.

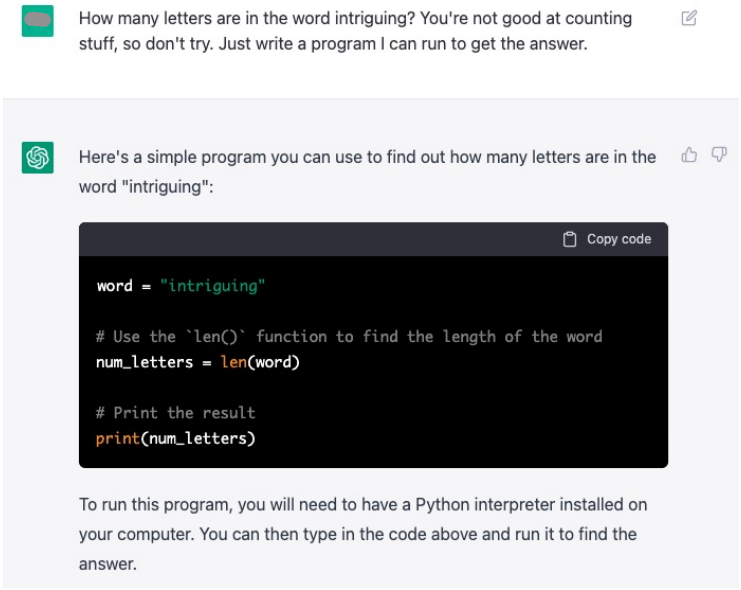
Figure 47: ChatGPT with PAL and CoT to answer a user-posted question




(a) Step-by-step reasoning struggle on counting the number of letters in the word "intriguing" which has ten letters.




(b) Explicitly instructing ChatGPT to reason step-by-step before generating answer still leads to the wrong answer.




(c) PAL takes a few lines of code and the execution could result in the correct answer.

Figure 48: ChatGPT with PAL and COT to answer a user-posted question

## H.7 Closer Look into Token-level Behaviors of Different Mechanisms

Beyond empirical results, we make initial attempts to gain a deeper understanding of the behavior of LLMs with different reasoning mechanisms by looking into the token-level log-likelihood of reasoning chains produced by CoT and PAL. We randomly selected 20 questions from the COLORED OBJECTS dataset, along with their corresponding CoT and PAL solutions. We then manually compared the two mechanisms by focusing on tokens with a low log-likelihood.

Our analysis reveals that CoT often has lower confidence in tokens related to numbers and quantitative information, the grounded position of spatial adjectives (e.g., right-most), properties such as the color of objects, and nouns that refer to the objects. Specifically, we found that this occurred in seven, six, two, and six examples out of the 20 we examined. In contrast, PAL uses list manipulations, such as `len(objects)`, and accesses objects and their associated properties through list indexing (e.g., `object[3][0]`). We found that the LLM is typically confident in producing these programs. Furthermore, we observed that while CoT requires different expressions for the same concept in different contexts, PAL almost always uses the same expression, which is presumably more robust. For example, when there are five objects, CoT predicts "the right-most thing is the fifth item on the list", and "the right-most thing is the third item on the list" when the number of objects is three. Occasionally, CoT also predicts "the right-most thing is last item on the list" which does not provide more concrete information. On the contrary, PAL confidently predicts `objects[-1]` consistently. The more consistent and uniform use of expressions in PAL can be attributed to the explicit and defined nature of programming languages, which allows for clear and accurate expressions.

## H.8 Datasets

In the following tables (Table 54,Table 55, Table 56), we presents statistics and examples for the datasets we considered.

| Dataset | N | Example |
|---|---|---|
| Reasoning about Colored Objects | 2000 | On the table, you see a bunch of objects arranged in a row: a purple paperclip, a pink stress ball, a brown keychain, a green scrunchiephone charger, a mauve fidget spinner, and a burgundy pen. What is the color of the object directly to the right of the stress ball? |
| Penguins in a Table | 149 | Here is a table where the first line is a header and each subsequent line is a penguin: name, age, height (cm), weight (kg) Louis, 7, 50, 11 Bernard, 5, 80, 13 Vincent, 9, 60, 11 Gwen, 8, 70, 15 For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm. We now add a penguin to the table: James, 12, 90, 12 How many penguins are less than 8 years old? |
| Date Understanding | 369 | 2015 is coming in 36 hours. What is the date one week from today in MM/DD/YYYY? |

Table 54: Reasoning datasets about everyday objects and concepts.

| Dataset | N | Example |
|---|---|---|
| Object Counting | 1000 | I have a chair, two potatoes, a cauliflower, a lettuce head, two tables, a cabbage, two onions, and three fridges. How many vegetables do I have? |
| Repeat Copy | 32 | Repeat the word duck four times, but halfway through also say quack. |

Table 55: Reasoning datasets about algorithmic problems.

| Dataset | N | Example |
|---|---|---|
| Math Reasoning [Cobbe et al., 2021] | 1319 | Olivia has $23. She bought five bagels for $3 each. How much money does she have left? |
| SVAMP [Patel et al., 2021] | 1000 | Each pack of dvds costs 76 dollars. If there is a discount of 25 dollars on each pack. How much do you have to pay to buy each pack? |
| ASDIV [Miao et al., 2020] | 2096 | Ellen has six more balls than Marin. Marin has nine balls. How many balls does Ellen have? |
| SINGLEOP [Koncel-Kedziorski et al., 2016] | 562 | If there are 7 bottle caps in a box and Linda puts 7 more bottle caps inside, how many bottle caps are in the box? |
| SINGLEEQ [Koncel-Kedziorski et al., 2016] | 508 | Benny bought a soft drink for 2 dollars and 5 candy bars. He spent a total of 27 dollars. How much did each candy bar cost? |
| ADDSUB [Koncel-Kedziorski et al., 2016] | 395 | There were 6 roses in the vase. Mary cut some roses from her flower garden. There are now 16 roses in the vase. How many roses did she cut? |
| MULTIARITH [Koncel-Kedziorski et al., 2016] | 600 | The school cafeteria ordered 42 red apples and 7 green apples for students lunches. But, if only 9 students wanted fruit, how many extra did the cafeteria end up with? |

Table 56: Examples from math reasoning datasets used in this work. Table from Wei et al. [2022c]. Datasets vary in number of words per question, with 47 words per question in Math Reasoning, 21 words in SINGLEOP, and ∼30 words in the other datasets.

| Question | CoT: GSM | CoT: GSMHard | Identical? |
|---|---|---|---|
| 90 single use contacts come in 1 box and will last Pete 45 days. Each box is $100.00 and currently 10% off. If he buys 2 boxes of contact, how much will each pair of contacts cost? | 90 contacts come in 1 box. Each box is $100.00. So 90 contacts cost $100.00. If he buys 2 boxes, then he will have 2 x 90 = 180 contacts. 180 contacts cost 2 x $100.00 = $200.00. So each pair of contacts costs $200.00 / 180 = $1.11. The answer is $1.11. | 90 single use contacts come in 1 box. So 1 box will last Pete 45 days. Each box is $5451440 and currently 10% off. So each box is $5451440 * 0.9 = $4906296. If he buys 2 boxes of contact, he will have 2 * 90 = 180 contacts. So each pair of contacts will cost $4906296 / 180 = $2725849. The answer is $2725849 | Yes |
| A chef bought 4 bags of onions. Each bag weighs 50 pounds. A pound of onions cost $1.50. How much did the chef spend? | The chef bought 4 bags of onions. Each bag weighs 50 pounds. So the total weight is 4 * 50 = 200 pounds. A pound of onions cost $1.50. So 200 pounds of onions cost 200 * 1.50 = 300 dollars. The answer is 300. | The chef bought 4 bags of onions. Each bag weighs 50 pounds. So the total weight is 4 * 50 = 200 pounds. A pound of onions cost $409201. So 200 pounds of onions cost 200 * 409201 = $81840200. The answer is $81840200. | Yes |
| A bag has a 5% discount. If it is marked $140, how much will you pay after the discount? | The discount is 5%. So you will pay 95% of the price. 95% of 140 is 133. The answer is 133. | The discount is 5%. So you will pay 95% of the price. 95% of $3921772.0722 is $3730654.0682. The answer is $3730654.0682 | Yes |
| A basket of green food costs $25 and a basket of red food costs $18. If you buy 3 baskets of green food and red food, how much will you have to pay in total if you get $2 off for each basket of red food? | 3 baskets of green food will cost 3 * 25 = 75 dollars. 3 baskets of red food will cost 3 * 18 = 54 dollars. But you get $2 off for each basket of red food. So you get 3 * 2 = 6 dollars off. So you pay 75 + 54 - 6 = 123 dollars. The answer is 123. | A basket of green food costs 9412763 dollars. A basket of red food costs 18 dollars. If you buy 3 baskets of green food and red food, you will have to pay 9412763 * 3 + 18 * 3 = 28238280 dollars. If you get 2 dollars off for each basket of red food, you will have to pay 28238280 - 18 * 2 = 28238244 dollars. The answer is 28238244 | No |
| A bumper car rink has 12 red cars. They have 2 fewer green cars than they have red cars. They have 3 times the number of blue cars as they have green cars. The rink also has yellow cars. If the rink has 75 cars in total how many yellow cars do they have? | There are 12 red cars. There are 2 fewer green cars than red cars. So there are 12 - 2 = 10 green cars. There are 3 times as many blue cars as green cars. So there are 3 * 10 = 30 blue cars. There are 12 + 10 + 30 = 52 cars that are not yellow. So there are 75 - 52 = 23 yellow cars. The answer is 23. | The rink has 9318738 red cars. They have 2 fewer green cars than red cars. So they have 9318738 - 2 = 9318736 green cars. They have 3 times the number of blue cars as green cars. So they have 3 * 9318736 = 27956208 blue cars. The total number of cars is 75. So the number of yellow cars is 75 - 9318738 - 9318736 - 27956208 = -55,828,829. The answer is -55,828,829 | Yes |

Table 57: Analysis of generations from CODEX. Manual analysis of 25 randomly generated thoughts reveals that 16 out of 25 thoughts were identical, whereas others were close paraphrases.

## H.9 Creating GSM-HARD

While replacing numbers in the question is easy using pattern matching, a more challenging aspect is recalculating the correct answer. Math Reasoning evaluation set contains 1319 samples, which is prohibitively expensive to perform manual re-calculation. Instead, we leverage PAL to assist obtaining the correct answers. For 71% of the examples where PAL is correct on Math Reasoning, we utilize the generated program and replace the initial value with the larger values. For example, if we create a harder version of the problem in Figure 8.3 by replacing $23 dollars with $15687 dollars, we correspondingly replace `money_initial=23` to `money_initial=15678`. Running the program could automatically produce the correct answer of the harder question. Notably, this annotation process assumes that a program that produces a correct answer to a Math Reasoning question indicates the correctness of the program itself. While this is not guaranteed due to possible spurious correlations, we manually checked 25 programs and found all of them are correct. For the incorrect 29% of the cases, we run PAL again and perform nucleus sampling [Holtzman et al., 2020] with temperature 0.7, and repeat the above process if any correct solution is found. Finally, the authors manually annotate the correct answer for 50 remaining cases that PAL was not able to solve after 100 iterations.

Note that the GSM-HARD benchmark was created automatically, it sometimes contains negative target values or target values that do not adhere to commonsense (e.g., *John bought 3.5 apples*). Unfortunately, we do not have the resources to manually annotate all examples, so our assumption is that there is a penalty of 5%-10% drop in performance for all models and prompting approaches that are evaluated on this benchmark. Since this penalty is similar to all approaches, we believe that the relative comparison between different approaches is the right thing to measure.

## H.10 GSM-HARD Analysis

Table 57 shows thoughts generated with CoT on Math Reasoning and GSM-HARD. A manual analysis reveals that a majority of the generated thoughts (16/25) were identical for Math Reasoning and GSM-HARD, indicating that larger numbers primarily diminish performance due to failure of LLM to do arithmetic..

## H.11 Prompts

We show here example PAL prompts we used for each data set. We show one example for each of the few-shot prompts. The fulls prompt can be found in our released code.

### Reasoning about Colored Objects

```
# Q: On the table, you see a bunch of objects arranged in a row: a
    purple paperclip, a pink stress ball, a brown keychain, a green
    scrunchiephone charger, a mauve fidget spinner, and a burgundy pen.
    What is the color of the object directly to the right of the stress
    ball?
# Put objects into a list to record ordering
objects = []
objects += [('paperclip', 'purple')] * 1
objects += [('stress ball', 'pink')] * 1
objects += [('keychain', 'brown')] * 1
objects += [('scrunchiephone charger', 'green')] * 1
objects += [('fidget spinner', 'mauve')] * 1
objects += [('pen', 'burgundy')] * 1
# Find the index of the stress ball
stress_ball_idx = None
for i, object in enumerate(objects):
    if object[0] == 'stress ball':
        stress_ball_idx = i
        break
# Find the directly right object
direct_right = objects[stress_ball_idx+1]
# Check the directly right object's color
direct_right_color = direct_right[1]
answer = direct_right_color
```

### Penguins in a Table

```
"""Q: Here is a table where the first line is a header and each
    subsequent line is a penguin: name, age, height (cm), weight (kg)
    Louis, 7, 50, 11 Bernard, 5, 80, 13 Vincent, 9, 60, 11 Gwen, 8, 70,
    15 For example: the age of Louis is 7, the weight of Gwen is 15 kg,
    the height of Bernard is 80 cm.  We now add a penguin to the table:
    James, 12, 90, 12
How many penguins are less than 8 years old?
"""
# Put the penguins into a list.
penguins = []
penguins.append(('Louis', 7, 50, 11))
penguins.append(('Bernard', 5, 80, 13))
penguins.append(('Vincent', 9, 60, 11))
penguins.append(('Gwen', 8, 70, 15))
# Add penguin James.
penguins.append(('James', 12, 90, 12))
# Find penguins under 8 years old.
penguins_under_8_years_old = [penguin for penguin in penguins if
    penguin[1] < 8]
# Count number of perguins under 8.
num_penguin_under_8 = len(penguins_under_8_years_old)
answer = num_penguin_under_8
```

Figure 50

### Date Understanding

```
# Q: 2015 is coming in 36 hours. What is the date one week from today in
    MM/DD/YYYY?
# If 2015 is coming in 36 hours, then today is 36 hours before.
today = datetime(2015, 1, 1) - relativedelta(hours=36)
# One week from today,
one_week_from_today = today + relativedelta(weeks=1)
# The answer formatted with %m/%d/%Y is
one_week_from_today.strftime('%m/%d/%Y')
```

**Math**

```
#Q: Olivia has \$23. She bought five bagels for \$3 each. How much money
    does she have left?
money_initial = 23
bagels = 5
bagel_cost = 3
money_spent = bagels * bagel_cost
money_left = money_initial - money_spent
print(money_left)

#Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On
    wednesday, he lost 2 more. How many golf balls did he have at the end
    of wednesday?
golf_balls_initial = 58
golf_balls_lost_tuesday = 23
golf_balls_lost_wednesday = 2
golf_balls_left = golf_balls_initial - golf_balls_lost_tuesday -
    golf_balls_lost_wednesday
print(golf_balls_left)

#Q: There were nine computers in the server room. Five more computers were
    installed each day, from monday to thursday. How many computers are now
    in the server room?

computers_initial = 9
computers_per_day = 5
num_days = 4  # 4 days between monday and thursday
computers_added = computers_per_day * num_days
computers_total = computers_initial + computers_added
print(computers_total)


#Q: If there are 3 cars in the parking lot and 2 more cars arrive, how many
    cars are in the parking lot?
cars_initial = 3
cars_arrived = 2
total_cars = cars_initial + cars_arrived
print(total_cars)

#Q: Leah had 32 chocolates and her sister had 42. If they ate 35, how many
    pieces do they have left in total?

leah_chocolates = 32
sister_chocolates = 42
total_chocolates = leah_chocolates + sister_chocolates
chocolates_eaten = 35
chocolates_left = total_chocolates - chocolates_eaten
print(chocolates_left)
```

Figure 52: Prompt used for mathematical reasoning (1/2)

```
#Q: Jason had 20 lollipops. He gave Denny some lollipops. Now Jason has 12
    lollipops. How many lollipops did Jason give to Denny?
jason_lollipops_initial = 20
jason_lollipops_after = 12
denny_lollipops = jason_lollipops_initial - jason_lollipops_after
print(denny_lollipops)



#Q: There are 15 trees in the grove. Grove workers will plant trees in the
    grove today. After they are done, there will be 21 trees. How many
    trees did the grove workers plant today?
trees_initial = 15
trees_after = 21
trees_added = trees_after - trees_initial
print(trees_added)


#Q: Shawn has five toys. For Christmas, he got two toys each from his mom
    and dad. How many toys does he have now?

toys_initial = 5
mom_toys = 2
dad_toys = 2
total_received = mom_toys + dad_toys
total_toys = toys_initial + total_received
print(total_toys)
```

Figure 53: Prompt used for mathematical reasoning (2/2)

### Object Counting

```
# Q: I have a chair, two potatoes, a cauliflower, a lettuce head, two
    tables, a cabbage, two onions, and three fridges. How many
    vegetables do I have?

# note: I'm not counting the chair, tables, or fridges
vegetables_to_count = {
    'potato': 2,
    'cauliflower': 1,
    'lettuce head': 1,
    'cabbage': 1,
    'onion': 2
}
print(sum(vegetables_to_count.values()))


# Q: I have a drum, a flute, a clarinet, a violin, four accordions, a
    piano, a trombone, and a trumpet. How many musical instruments do I
    have?

musical_instruments_to_count = {
    'drum': 1,
    'flute': 1,
    'clarinet': 1,
    'violin': 1,
    'accordion': 4,
    'piano': 1,
    'trombone': 1,
    'trumpet': 1
}
print(sum(musical_instruments_to_count.values()))




# Q: I have a chair, two ovens, and three tables. How many objects do I
    have?

objects_to_count = {
    'chair': 1,
    'oven': 2,
    'table': 3
}
print(sum(objects_to_count.values()))
```

Figure 54: Prompt used for OBJECT COUNTING.

**Repeat Copy**

```
# Q: Repeat the word duck four times, but halfway through also say quack

result = []
for i in range(1, 5):
    result.append("duck")
    if i == 2:
        result.append("quack")
print(" ".join(result))


# Q: Print boolean eleven times, but after the 3rd and 8th also say
    correct

result = []
for i in range(1, 12):
    result.append("boolean")
    if i == 3 or i == 8:
        result.append("correct")
print(" ".join(result))

# Q: say java twice and data once, and then repeat all of this three
    times.

result = []
tmp = ["java", "java", "data"]
for i in range(3):
    result.extend(tmp)
print(" ".join(result))


# Q: ask a group of insects in what family? four times. after the fourth
    time say The happy family

result = []
tmp = []
for i in range(1, 5):
    tmp.append("a group of insects in what family?")
tmp.append("The happy family")
result.extend(tmp)
print(" ".join(result))
```

Figure 55: Prompt used for REPEAT COPY.

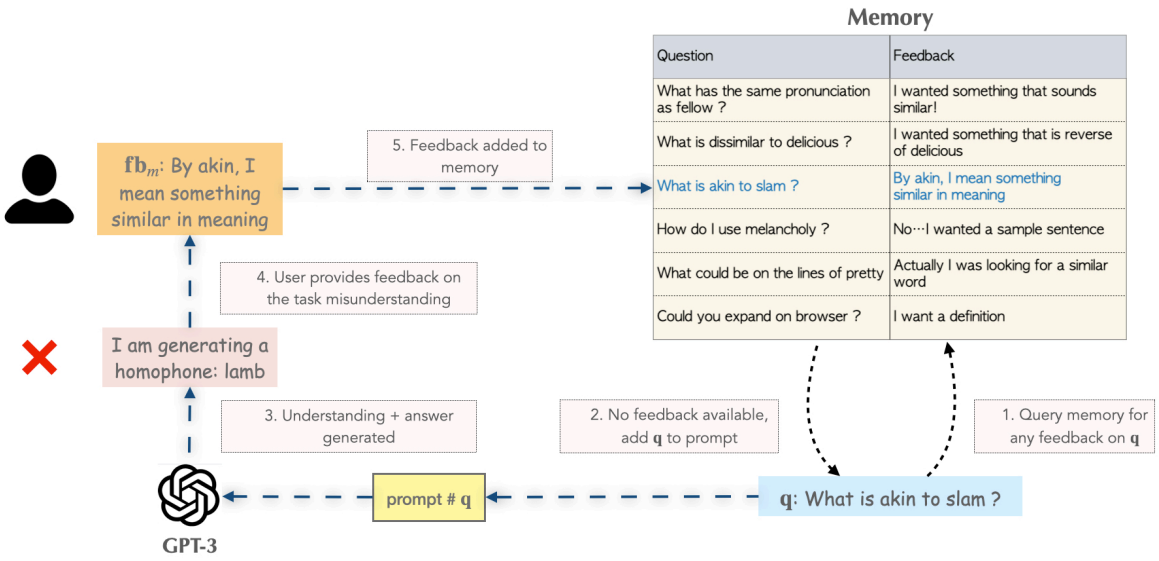



Figure 56: MemPrompt: adding to memory. User enters a question for which no feedback is available (steps 1, 2). Directly prompting GPT-3 with the question leads to incorrect answer and understanding (step 3). User-provides feedback on the incorrect understanding (step 4), which is added to memory (step 5).

# I Chapter 9: Memory-assisted Prompting

## I.1 Inside MemPrompt: Populating and using the memory

MemPrompt maintains a growing memory of recorded cases where a feedback was provided to clarify the user's misunderstood intent. This flow is presented in Figure 56 that shows a sequence of steps 1-5 on how the memory is populated.

MemPrompt also produces enhanced prompts for any new query based on the user feedback on similar cases recorded previously in the memory. Figure 57 presents the sequence of steps 1-3 involved in retrieving and applying a past feedback on a similar case.

## I.2 Generative IR (GUD-IR)

*A note on feedback and understanding* Feedback $\mathbf{fb}$ and understanding a are two concepts that we repeatedly use in this work. Briefly, MemPrompt requires a model to spell out its understanding of the instruction (a). The user can then provide a feedback $\mathbf{fb}$ on the understanding. In the prompt, both $\mathbf{fb}$ and a are identical. Such examples are of the form $\mathbf{x}$, a $\rightarrow$ a, $\mathbf{y}$ and their main purpose is to reinforce that model the input feedback a be used to generate the output.

### Introduction

One of the key strengths of MemPrompt is its ability to leverage feedback provided on earlier inputs $\mathbf{x}$ to improve a current input. This is achieved by retrieving a feedback from memory $\mathcal{M}$ using $\mathbf{x}$ as the key. An underlying assumption of this process is that similar inputs will admit

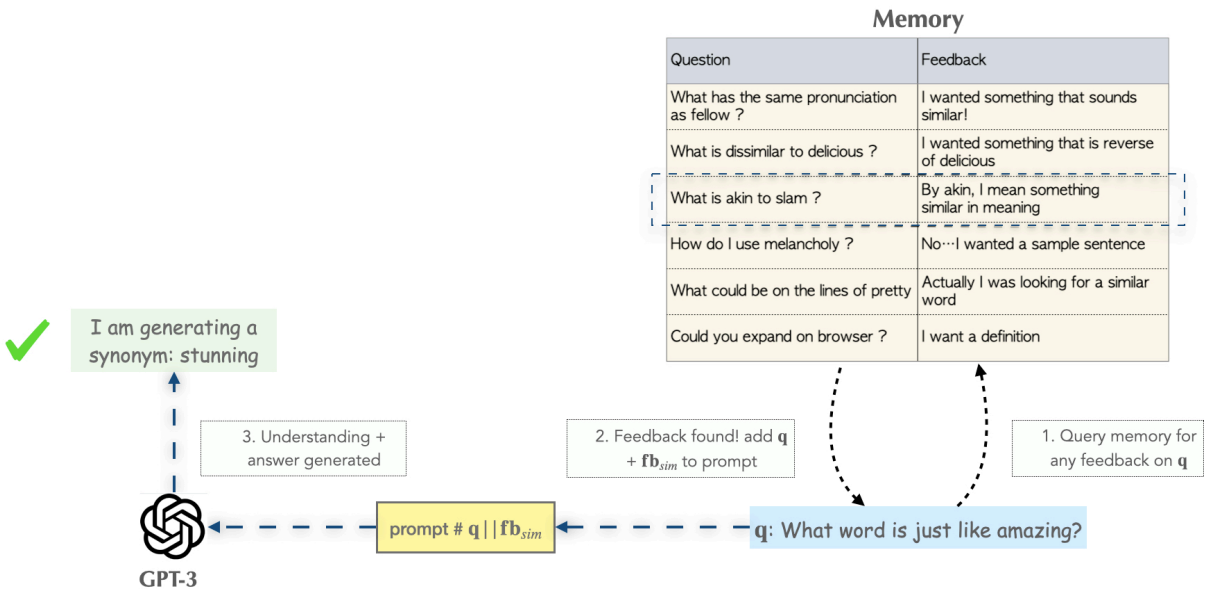



Figure 57: MemPrompt: retrieving feedback from memory. User enters a question which GPT-3 has incorrectly answered in the past, and has received feedback from a user (step 1). The feedback is retrieved from memory (step 2), and both question and feedback are added to the prompt. The prompt contains examples that allow GPT-3 to react to user feedback and generate correct understanding and answer.

similar feedback, allowing us to use the feedback provided for one situation on another. For two input situations $s_i$ and $s_j$ with respective feedback $fb_i$ and $fb_j$, this assumption can be succinctly stated as:

$$s_i \sim s_j \implies fb_i \sim fb_j$$

The ethical reasoning dataset with natural language feedback, ERT-NL, provides a unique challenge for this assumption because lexically dissimilar situations might have the same feedback. As a concrete example, consider an input situation $s_i$: *tom hated skating because he had no sense of balance* – with a feedback $fb_i$: *this question is about practicing more when you want to improve your skills*. Suppose that our system has already seen $s_i$ and has received a feedback $fb_i$ (i.e., there is an entry in $\mathcal{M}$: $s_i \rightarrow fb_i$). Next, suppose a user enters a new situation $s_j$: *jordyn was trying to improve her soccer skills*. As usual, MemPrompt will try to retrieve feedback for a *similar* situation. However, such retrieval is going to be challenging, because $s_i$ (*tom hated skating because he had no sense of balance*) has little to no overlap with $s_j$ (*jordyn was trying to improve her soccer skills*), although humans can easily tell that both situations are about improving skills. Consequently, MemPrompt may fail to retrieve the relevant feedback $fb_i$ or worse, may retrieve a misleading feedback.

The fact that two ostensibly dissimilar inputs two inputs $(\mathbf{x}_i, \mathbf{x}_j)$ may share the same feedback makes vanilla retrieval non-viable for our setting. We deal with this challenging situation with two different solutions of increasing complexity.

**Initial approach: Learning a feedback similarity function**

Since the surface level similarity of input situations is not enough to capture similarity of respective feedback, we attempt to learn a function $f_\theta$ that will map similar inputs $\mathbf{x}_i$ and $\mathbf{x}_j$ to similar representations if the corresponding feedback $fb_i$ and $fb_j$ are close to each other, and vice-versa. A natural choice is training an embedding function $f : \mathbf{x} \rightarrow \mathrm{R}^d$ supervised by $\texttt{cos}(fb_i, fb_j)$ where $\texttt{cos}$ is the cosine similarity ($\texttt{cos}(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a}^T \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}$). Thus, the objective function is:

$$\mathcal{L}_\theta = (\texttt{cos}(f_\theta(\mathbf{x}_i), f_\theta(\mathbf{x}_j)) - \texttt{cos}(fb_i, fb_j))^2$$

Intuitively, this objective function will encourage the similarity between the inputs ($\texttt{cos}(f_\theta(\mathbf{x}_i), f_\theta(\mathbf{x}_j))$) to be high when the corresponding feedback are similar, and vice-versa.

Feedback retrieval proceeds as follows: an input $s_i$ is embedded using $f_\theta$, and $f_\theta(s_i)$ is then used to retrieve a feedback from the memory, with the hope that representations $f_\theta(s_i)$ and $f_\theta(s_j)$ will be similar after the training.

While in principle this objective function should be enough to learn informative representations that bring two inputs with similar feedback close, we found the training to be unstable. We attribute this to the fact that two extremely dissimilar situations can have identical feedback. Given limited training data, it might be unrealistic to train similarity functions that can capture all possible cases where the same feedback applies to two situations. As a way to circumvent this, we also experiment with a generative version of our method, described next.

$\mathbf{x}$

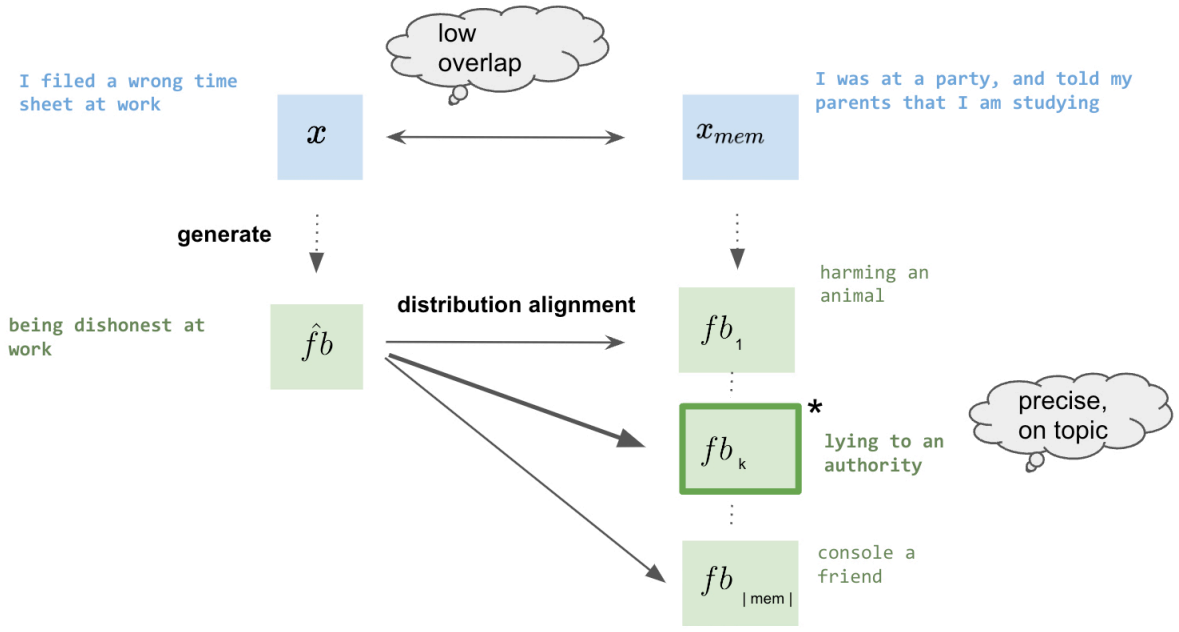



Figure 58: **Overview of GUD-IR.**

**Proposed approach: Training generative model for retrieving similar feedback**

To address these retrieval issues, we propose **GUD-IR** (Generated UnDerstanding for explainable IR). The key intuition for our approach relies on substituting $f_\theta : \mathbf{x} \rightarrow \mathrm{R}^d$ (latent space projection) with $f_\theta : \mathbf{x} \rightarrow fb$ (generated understanding of $\mathbf{x}$). Concretely, instead of learning a function that maps a question to a $d$ dimensional vector, we train a generative model that directly maps an input to a rough understanding. The generated rough understanding is then used as a key to retrieve a relevant understanding from the database using any off-the-shelf retrieval method. This two-step *generate-then-retrieve* procedure has benefits: (i) it alleviates sparsity issues that we found latent space projection methods were unable to deal with[12] (ii) the overall retrieval becomes explainable and debuggable.

Our approach is inspired and supported by the recent success of generate and retrieve Mao et al. [2021] methods. However, despite the similarity, the methods have different goals: Mao et al. [2021] leverage generative models for query expansion, whereas our goal is explainable input understanding. Moreover, their implementation is geared towards open-domain QA, while ours is towards explainable input understanding. Thus, it is non-trivial to adapt similar ideas to our tasks effectively.

Specifically, we train a SEQ2SEQ model, (e.g., T5 [Raffel et al., 2020c]), that maps each input $\mathbf{x}$ to a corresponding output $\mathbf{fb}$. The feedback is now retrieved in a two step process:

1. The generative model $f_\theta$ is used to generate a noisy feedback for $s_i$, $\hat{fb}$.
2. $\hat{fb}$ is used as a key to *search* over the set of already present feedbacks, to retrieve the nearest one.

Instead of directly using clarification to lookup the nearest feedback, we first transform the input to the space of clarifications, then search over the set of already present clarifications. Figure 58 presents an overview of our *generation then reshape* approach (GUD-IR). As we discuss in Section 9.3.1, GUD-IR was key to achieving good performance for the ERT-NL task.

In addition to the task accuracy, we plot the distribution of $\texttt{sim}(\hat{u}, \hat{u}^*)$ (similarity of the true and retreived feedback) over the test set for different retrieval methods. Figure 59 shows this distribution using GUD-IR and using surface-level similarities. The probability mass shifts towards a higher similarity range for GUD-IR.

The lexical reasoning and WEBQA tasks present a simpler setting for retrieval, as similarity of keys indicates a similarity of values. For such cases, we use Sentence transformers [Reimers and Gurevych, 2019] to encode the query, and cosine similarity with a threshold of 0.9 to find a matching key.

## I.3 Querying GPT-3-175B using OpenAI API

We use the OpenAI API for querying GPT-3-175B.[13] The python code is listed below. Here, "PROMPT" is set to prompt shown in I.4, followed by the input question $\mathbf{x}$ and feedback $\mathbf{fb}$ if

[12] e.g., there are only eight popular emotions but can lead to a large number of diverse situations. Hence, many inputs can map to the same principle $\mathbf{fb}$. This mapping becomes increasingly difficult for a model as the specificity of $\mathbf{fb}$ increases, because of sparsity issues. This is exacerbated when the input situations are diverse and previously unseen.

[13] https://beta.openai.com/docs/introduction, we use 'text-davinci-001'

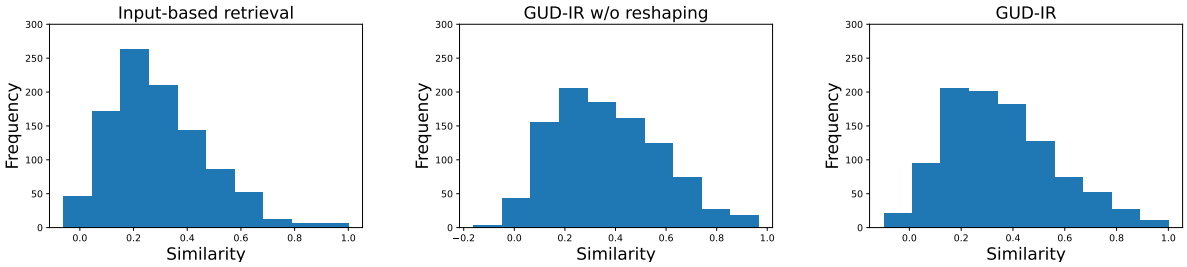



Figure 59: Distribution of similarity scores between expected $\mathbf{fb}^*$ and $\hat{u}$ for retrieval (left) and GUD-IR (right). The similarity scores are higher using GUD-IR.

applicable.

We used a temperature of 0.0 for factual QA (WEBQA) experiments to select the most likely token at each step, and this setting does not require generating diverse answers, as one would expect for a factual domain. For ERT-CAT and ERT-NL, we found that a higher temperature ($\sim$ 0.7) was causing a large divergence in the performance (a difference of $\pm 10\%$ accuracy across runs), making reproducibility challenging – similar observations were made by Summers-Stay et al. [2021]. Thus, we used to a temperature of 0.0 for ERT experiments. A temperature of 0.7 was used for all the other experiments.

```
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")

response = openai.Completion.create(
  engine="davinci",
  prompt="PROMPT",
  temperature=0.7,
  max_tokens=64,
  top_p=1,
  frequency_penalty=0,
  presence_penalty=0
)
```

## I.4 Prompt

GPT3 is queried using a prompt $p$ of example i/o behaviors, followed by the actual question $\mathbf{x}$ and (optionally) retrieved feedback $\mathbf{fb}$. It then generates the understood intent a and answer $\mathbf{y}$ as a continuation. a and $\mathbf{y}$ are expressed a single sentence, e.g., "[The synonym for ¡word¿ is] [¡word¿]" Figure 60 shows this prompt $p$, containing a mixture of $(\mathbf{x} \rightarrow \mathrm{a}, \mathbf{y})$ and $(\mathbf{x}, fb \rightarrow \mathrm{a}, \mathbf{y})$ "training" tuples.

What is the homophone for ¡ wring ¿ ?
#
the homophone for wring is ring END
#
how do you use ¡ highway ¿ in a sentence?
#
a sentence with highway is: soldiers stand guard along the [ highway ] END
#
What is the synonym for ¡ surrogate ¿ ?
#
the synonym for surrogate is substitute END
#
how do i use ¡ fog ¿ ? — clarification: when i ask for how do i use, i want a sentence.
#
a sentence with fog is: a rising sun burns the [ fog ] off a city END
#
What sounds like ¡ sighted ¿ ? — clarification: when I ask for sounds like, I want a homophone.
#
the homophone for sighted is cited END
#
what is like ¡ provident ¿ ? — clarification: when I ask for like, I want a synonym.
#
the synonym for provident is prudent END
#
can you define ¡ rider ¿ ? — clarification: when i ask for define, i want a definition.
#
the definition of rider is a person who is riding something. END
#
What is the opposite of ¡ citation ¿ ? — clarification: when I ask for opposite, I want an antonym.
#
the antonym for citation is award END

Figure 60: The prompt used for our tasks. During inference, an input question $\mathbf{x}_i$, and optionally a feedback $fb_i$ is appended after this prompt, and the model is expected to generate the answer $\mathbf{y}_i$ and its understanding of the question intent $\mathrm{a}_i$ as a continuation. The prompt contains examples of the form $(\mathbf{x} \rightarrow \mathrm{a}, \mathbf{y})$, expressed "$\mathbf{x}$ # a $\mathbf{y}$ END #", and $(\mathbf{x}, fb \rightarrow \mathrm{a}, \mathbf{y})$, expressed "$\mathbf{x}$ — clarification: $\mathbf{fb}$ # a $\mathbf{y}$ END #". (a and $\mathbf{y}$ are expressed together as a single sentence, e.g., "[The synonym for ¡word¿ is] [¡word¿].")

Find the right word after removing random letters from ¡ t!r/e/a/s/u/r.e!s ¿
#
the word after removing symbols from t!r/e/a/s/u/r.e!s is treasures END
#
Find the original word after ignoring the punctuation and spaces in ¡ e ¿
#
the word after removing symbols from e is elders END
#
Find the right word given this cycled word: ¡ lprovisiona ¿ ?
#
the uncycled version of lprovisiona is provisional END
#
Make a word while keeping the first and last char ¡ vosiin ¿ ?
#
the anagram 1 for vosiin is vision END
#
Find the original word that is interspersed in ¡ f.i.n!e/p.i/x ¿
#
the word after removing symbols from f.i.n!e/p.i/x is finepix END
#
Find the right word given this rotated word: ¡ ibitioninh ¿ ?
#
the uncycled version of ibitioninh is inhibition END
#
Figure out the word which has the same first two and the last two char ¡ watsed ¿ ? — clarification: when I want you to figure out the word which has the same first two and the last two char, I mean anagram 2.
#
the anagram 2 for watsed is wasted END
#
Make a word while keeping the first and last char ¡ isucnase ¿ ? — clarification: when I want you to make a word while keeping the first and last char, I mean anagram 1.
#
the anagram 1 for isucnase is issuance END
#
Unscramble everything except the first two and the last two char ¡ acotrs ¿ ? — clarification: when I want you to unscramble everything except the first two and the last two char, I mean anagram 2.
#
the anagram 2 for acotrs is actors END

Figure 61: The prompt used for our word scrambling tasks derived from Brown et al. [2020a]

**Situation:** last day of high school food fight and I nailed a special needs girl square in the face with a heavy muffin.
**Morality Judgment:** This question is about: getting into food fights with people. The answer is It's bad. END
#
**Situation:** when phone shopping at stores, I used to write inappropriate stuff and post it to random peoples book of faces accounts who would forget to logoff on the display phones. This question is about: messing with people's social media accounts.
**Morality Judgment:** This question is about: messing with people's social media accounts. The answer is It's bad. END
#
**Situation:** i had an uncomfortable talk with my 10yo about puberty and feel like avoiding him for a few days.
**Morality Judgment:** This question is about: being honest with your kids. The answer is It's good. END
#
**Situation:** i heard once that if you cringe about your past it just means you've grown as a person, for anyone who needs to hear it. This question is about: regretting poor decisions from your past.
**Morality Judgment:** This question is about: regretting poor decisions from your past. The answer is It's okay. END
#
**Situation:** my cat hopped up in my lap, cleaned herself and went to sleep. now I'm a captive and I want to move but I can't. This question is about: waking up an animal because you would like to move.
**Morality Judgment:** This question is about: waking up an animal because you would like to move. The answer is It's okay. END
#
**Situation:** if it wasn't for the pain it would cause to my family, I would have thrown myself off a bridge a while ago.
**Morality Judgment:** This question is about: caring about your family's feelings. The answer is It's good. END
#

Figure 62: The prompt used for ERT-NL task. The user provides a situation and the model generates a morality judgement. In addition to the morality judgment, the model also provides a high-level understanding of what the task is about.

**Situation**: i am selectively racist towards ”generic” indian recruiters because they waste my time with absolute zero results.
**Morality Judgment**: This question is about: trating you differently because of your race. The answer is It’s bad. END
#
**Situation**: I’ve misunderstood the phrase ‘when the going gets tough the tough get going’ for my entire life. This question is about: not understanding cliches when young.
**Morality Judgment**: This question is about: not understanding cliches when young. The answer is It’s okay. END
#
**Situation**: when phone shopping at stores, I used to write inappropriate stuff and post it to random peoples book of faces accounts who would forget to logoff on the display phones. This question is about: messing with people’s social media accounts.
**Morality Judgment**: This question is about: messing with people’s social media accounts. The answer is It’s bad. END
#
**Situation**: i had an uncomfortable talk with my 10yo about puberty and feel like avoiding him for a few days.
**Morality Judgment**: This question is about: being honest with your kids. The answer is It’s good. END
#
**Situation**: i heard once that if you cringe about your past it just means you’ve grown as a person, for anyone who needs to hear ito. This question is about: regretting poor decisions from your past.
**Morality Judgment**: This question is about: regretting poor decisions from your past. The answer is It’s okay. END
#
**Situation**: my cat hopped up in my lap, cleaned herself and went to sleep. now I’m a captive and I want to move but I can’t. This question is about: waking up an animal because you would like to move.
**Morality Judgment**: This question is about: waking up an animal because you would like to move. The answer is It’s okay. END
#
**Situation**: if it wasn’t for the pain it would cause to my family, I would have thrown myself off a bridge a while ago.
**Morality Judgment**: This question is about: caring about your family’s feelings. The answer is It’s good. END

Figure 63: The prompt used for ERT-CAT task. The user provides a situation and the model generates a morality judgement. In addition to the morality judgment, the model also provides a high-level understanding of what the task is about.

## I.5 Datasets for lexical question-answering tasks

As mentioned in Section 9.3, we focus on five different linguistic QA tasks. The source of data for each of these tasks is listed below:

1. The synonyms (syn) and antonyms (ant) were obtained from Nguyen et al. [2016].[14]
2. The homophones (hom) were obtained using homz https://github.com/cameronehrlich/homz. We use the closest homophone returned by homz for each word in the English dictionary.
3. The definitions (defn) were sourced from *The Online Plain Text English Dictionary* https://github.com/eddydn/DictionaryDatabase
4. Examples for usage in a sentence (sent) are from Commongen Lin et al. [2020b].

### Templates

We manually created 15 task templates with three variants of phrasing the question for each task. The data (word1, word2) in the code is initialized with the entries in the four sources mentioned above. The complete template file is available in the project repository https://github.com/madaan/memprompt/tree/main/src/templates.

### Sample questions

Tables 62, 63, and 63 list some sample **x**-**y** for settings where the question was asked as a linguistic variation, in Hindi, and in Punjabi, respectively.

## I.6 MemPrompt with label feedback

Our current approach requires the model to verbalize its understanding of the question, on which a user provides feedback. Such a setup might not be possible, for instance, due to the nature of questions. Can MemPrompt be effectively used in such settings as well? To investigate this, we experiment with factual question answering on the WEBQA dataset [Berant et al., 2013], and use the test set provided by Berant et al. [2013] for all experiments (2032 questions). The WEBQA dataset consists of factual questions (*which language is spoken in Canada?*) with multiple answers (*English, French*), and is a popular dataset for benchmarking the performance of GPT-3 on question answering in a few-context prompting setup.

**Inference** Let $k$ be the number of examples (i.e., question-answer pairs) in the prompt. For a given question $q$, We keep half ($k/2$) examples fixed in the prompt, whereas the other half $k/2$ examples are retrieved from a memory of feedback $M$. As before, on receiving a question $q$, consults a memory $M$ to see if a similar question has been asked before. However, different from earlier setups, in this case, we retrieve $k/2$ most similar questions from the memory $M$ on **which the system has been wrong earlier**. The corresponding true answers are also retrieved. These

[14] https://www.ims.uni-stuttgart.de/en/research/resources/experiment-data/lexical-contrast-dataset/

| % | Question ($q$) | Example 1 | Example 2 | Example 3 |
|---|---|---|---|---|
| 1 | what highschool did harper lee go to? | what did st augustine do? | who is keyshia cole dad? | when did charles goodyear invented rubber? |
| 4 | what college did albert einstein go to? | what highschool did harper lee go to? | who did tim tebow play college football for? | what timezone is utah in? |
| 40 | where did john mayer go to college? | what school did michael jackson go to high school? | where did derek fisher go to college? | what style of music does john mayer play? |
| 90 | where did scott fitzgerald go to college? | who was f. scott fitzgerald? | where did otto frank go to college? | where did derek fisher go to college? |

Table 58: Relevant examples fetched with time: as time proceeds, the examples fetched from the memory become increasingly relevant to the input question, leading to increasingly accurate predictions.

| Timestep | Question | Neighbor |
|---|---|---|
| 12 | is not wanting a dirty person harassing you morally ok? | is not wanting to hear foul language morally good? |
| 70 | when adam decided to stop living a lie and announces he's gay, are you expected to pretend you're straight when you're gay? | are you expected to pretend you're straight when you're gay? |
| 85 | Should I help someone when they are having trouble? | must you intervene if you see someone taking advantage of another disabled person? |
| 230 | needing a hug to feel good. | is wanting to cuddle with loved ones okay? |

Table 59: Relevant examples retrieved at increasing timesteps: as time proceeds, the examples fetched from the memory become relevant to the input question, leading to accurate predictions.

$k/2$ retrieved question-answer pairs are combined with the $k/2$ fixed questions to create a prompt, and query GPT-3. Let $a'$ be the generated answer.

**Growing memory of errors** $M$ In our setup, we assume an expert user (or a teacher) that knows the true answer $a$ for a given query $q$. The expert user compares the GPT-3 generated answer $a'$ with $a$. If the generated answer is correct ($a' = a$), no further action is taken. If not, the entry $((q, a))$ is added to the memory $M$. As time passes, $M$ is populated with an increasing number of challenging examples that the model has been wrong on. Thus, the retrieved $k/2$ examples get more relevant with time, aiding the accuracy. In the experiments, we set $k = 16$ due to budget constraints (note that the setups used in Liu et al. [2022a] and Brown et al. [2020a] set $k = 64$, but their results are comparable to our baseline with $k = 16$).

**Results** Similar to ERT and word reasoning tasks, a memory of errors helps in increasing accuracy with time over 3,000 points in the test split of the WEBQA dataset (Figure 64). This is expected, as $M$ gathers more examples on which GPT-3-175B has been wrong before. Adding these examples in the prompt avoids the model in repeating these mistakes.

To check if examples that belong to a similar domain improve with time, we cluster the questions in the test set of WEBQA, and randomly select three clusters for our analysis. Table 60 shows the top three of the 8 ($k = 16/2$) examples retrieved from $M$ for the *alma mater* cluster.[15] All of these questions relate to the alma mater of famous personalities. As the inference begins (with an empty $M$), the examples are not relevant to $q$. However, towards the end, almost all the samples are relevant to the given question.

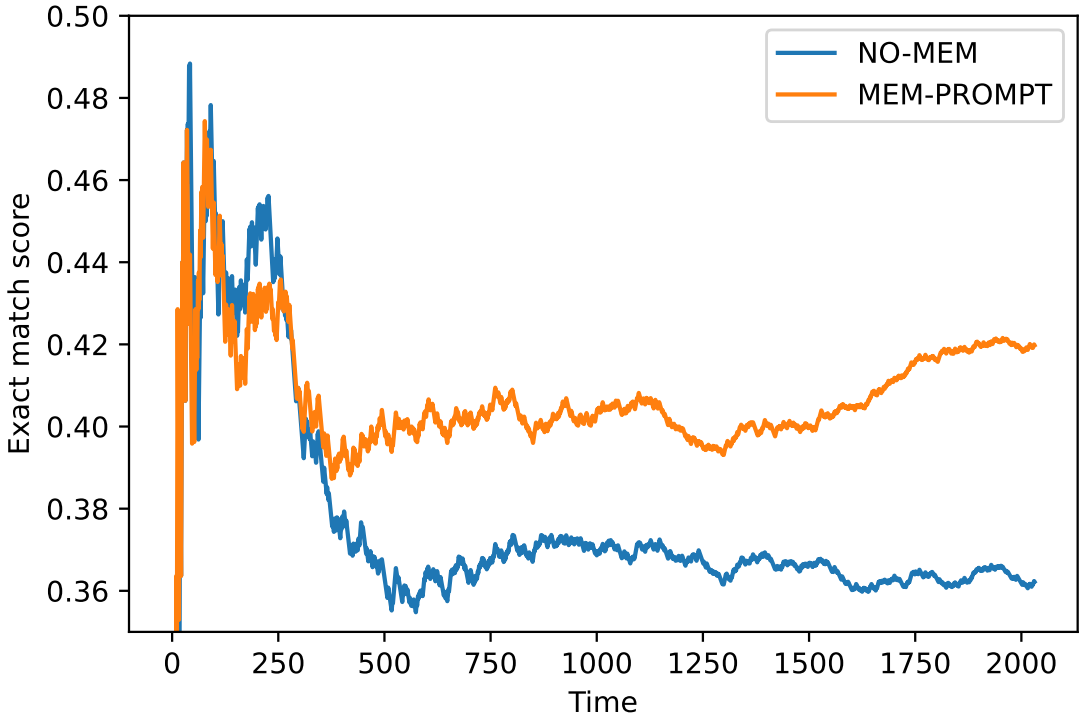



Figure 64: Instruction accuracy vs. time for WEBQA.

**Factual question answering Examples**

Tables 60 and 61 show additional examples for questions from WEBQA which get additionally relevant examples as time proceeds. The examples include questions that belong to the domains of Alma mater, Soccer, and Language.

## I.7 Finding similar questions in low-resource settings

We also experimented using queries in Hindi and Punjabi, with (English) feedback clarifying the queries' intent when GPT3 predictably misunderstands the task.Figure 65 confirms significant gains using memory in this OOV setting. This setup highlights the case when the user does not

[15]Additional examples are included in Appendix I.6.

| Domain | % Finished | Question | Neighbor 1 | Neighbor 2 | Neighbor 3 |
|---|---|---|---|---|---|
| Alma mater | 1 | what high-school did harper lee go to? | what did st augustine do? | who is keyshia cole dad? | when did charles goodyear invented rubber? |
| Alma mater | 5 | what college did albert einstein go to? | what high-school did harper lee go to? | who did tim tebow play college football for? | what timezone is utah in? |
| Alma mater | 75 | where did john steinbeck go to college? | where did john mayer go to college? | what college did john stockton go to? | where did otto frank go to college? |
| Alma mater | 95 | where did scott fitzgerald go to college? | who was f. scott fitzgerald? | where did otto frank go to college? | where did derek fisher go to college? |
| Soccer | 33 | who did nasri play for before arsenal? | what year did ray allen join the nba? | who does donnie wahlberg play in the sixth sense? | what does david beckham play? |
| Soccer | 65 | who has pudge rodriguez played for? | who does nolan ryan play for? | who did carlos boozer play for? | who does ronaldinho play for now 2011? |
| Soccer | 99 | what team did david beckham play for before la galaxy? | who does david beckham play for? | what does david beckham play? | what team does david beckham play for in 2012? |

Table 60: Relevant examples retrieved for WEBQA QA task (Section I.6). The retrieved examples get increasingly relevant as time proceeds.

| Domain | % Finished | Question | Neighbor 1 | Neighbor 2 | Neighbor 3 |
|---|---|---|---|---|---|
| Language | 1 | what does jamaican people speak? | when was ancient egypt created? | where is the denver broncos stadium located? | what is the name of the capital of spain? |
| Language | 20 | what are the two official languages of paraguay? | what do portuguese people speak? | what language does cuba speak? | where is mission san buenaventura located? |
| Language | 37 | what language does colombia? | what language does cuba speak? | what was the first language spoken in spain? | what is serbian language called? |
| Language | 85 | what language does peru speak? | what are the official languages of the eu? | where is the latin language from? | what do portuguese people speak? |
| Language | 90 | what language do they speak in colombia south america? | how many languages do they speak in spain? | where is the latin language from? | what language does cuba speak? |

Table 61: Relevant examples retrieved for WEBQA QA task (Section I.6). The retrieved examples get increasingly relevant as time proceeds.

speak fluent English and uses mixed language code, e.g., transcription in English and mixing words from another language to ask questions.

In low-resource settings (e.g., queries in transcribed Punjabi or Hindi), we perform similarity matching between a given question and a question in the memory by using surface-form similarity. Specifically, we use Levenshtein distance to determine the closest query in the memory. We note that as the memory grows large, we can use mechanisms such as FAISS [Johnson et al., 2019b] for trained memory, and suffix-trees for fast retrieval using surface form similarity.

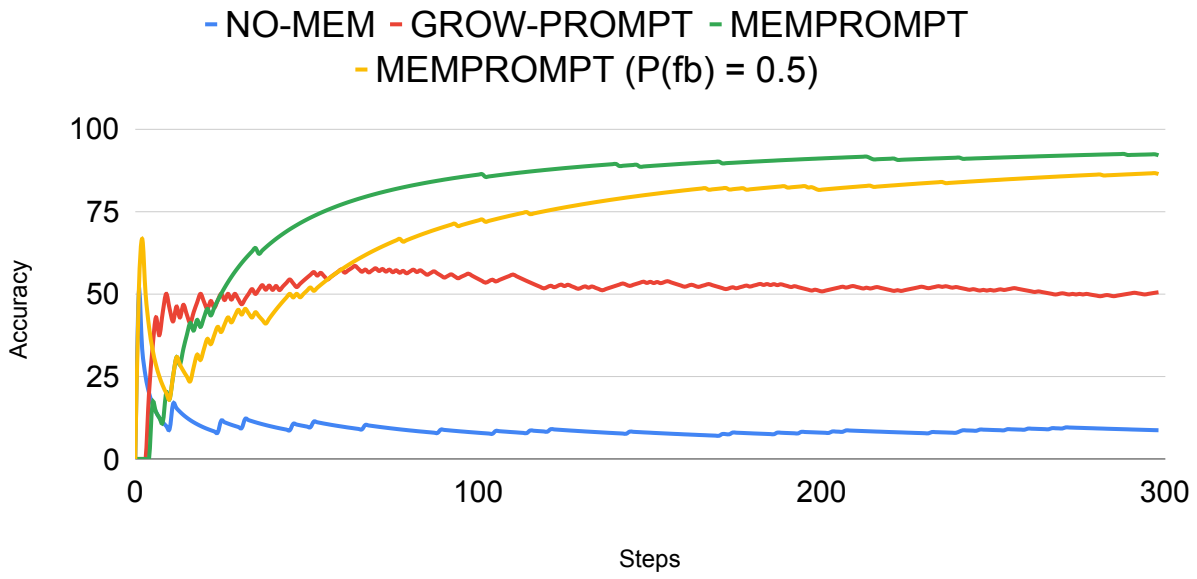



Figure 65: **Finding 2** Large gains on queries asked in English and Punjabi by MemPrompt.

## I.8 Sample results

Table 65 shows randomly sampled x-y pairs, and the corresponding y generated by GPT-3-175B and MemPrompt. The complete set of outputs is located in the project repository https://github.com/madaan/memprompt/tree/main/results.

| **Question (x)** | **Answer (y)** | type |
|---|---|---|
| What is the opposite of ¡ misconstrue ¿ ? | the antonym for misconstrue is verify | ant |
| What is the opposite of ¡ gross ¿ ? | the antonym for gross is polite | ant |
| expand on ¡ chelicera ¿ ? | the definition of chelicera is One of the anterior pair of mouth organs | defn |
| what is a sentence that can be used to define ¡ mawseed ¿ ? | the definition of mawseed is The seed of the opium poppy. | defn |
| what has a ¡ bitt ¿ like ring to it ? | the homophone for bitt is bit | hom |
| what can one confuse with ¡ holed ¿ ? | the homophone for holed is hold | hom |
| ¡ spread ¿ can be used how ? | a sentence with spread is: a couple of sheep are spread out in a field | sent |
| make something with ¡ pot ¿ ? | a sentence with pot is: bonsai tree in pot at zen garden . | sent |
| What is akin to ¡ musician ¿ ? | the synonym for musician is instrumentalist | syn |
| What is akin to ¡ zigzag ¿ ? | the synonym for zigzag is move | syn |

Table 62: Sample x-y pairs in English. The same type of question can be asked in multiple ways. Our method makes no assumptions as to how a question might be asked.

# J Chapter 10: Self-Refine

## J.1 Evaluation Tasks

Table 66 lists the tasks in our evaluation, and examples from each task.

| **Question (x)** | **Answer (y)** | type |
|---|---|---|
| ( tabulate ) ka ulta kya hai ? | the antonym for tabulate is randomize | ant |
| ( foot ) ka vilom kya hai ? | the antonym for foot is head | ant |
| ( lettish ) ka matlab kya hota hai ? | the definition of lettish is The language spoken by the Letts. See Lettic. | defn |
| ( housing ) ka arth kya hai ? | the definition of housing is An appendage to the hames or collar of a harness. | defn |
| sunne mai ( perl ) jaisa kya hai ? | the homophone for perl is pearl | hom |
| ( council ) jaisa kya sunai deta hai ? | the homophone for council is conceal | hom |
| ( city ) ko ek vakya mai kaise likhen ? | a sentence with city is: the city takes on an even more interesting hue during event | sent |
| ( fly ) ko ek vakya mai kaise likhen ? | a sentence with fly is: airplane fly into a storm cloud | sent |
| ( critique ) kai samaan kya hota hai ? | the synonym for critique is evaluate | syn |
| ( psychiatric ) kai samaan kya hota hai ? | the synonym for psychiatric is mental | syn |

Table 63: Sample x-y pairs in Hindi.

| **Question (x)** | **Answer (y)** | type |
|---|---|---|
| ( edit ) de ult ki hunda ae ? | the antonym for edit is expand | ant |
| ( spring ) ton bhin ki ae ? | the antonym for spring is compress | ant |
| ( patricide ) di paribhasha dasso ? | the definition of patricide is The crime of one who murders his father. Same as Parricide. | defn |
| ( sankhya ) da matlab ki hunda ae ? | the definition of sankhya is A Hindoo system of philosophy which refers all things to soul and a rootless germ called prakriti | defn |
| sunnan vich ( site ) varga ki ae ? | the homophone for site is sites | hom |
| ( chance ) da samnam ki ae ? | the homophone for chance is chants | hom |
| ( enter ) nu ek vak vich kidan vartiye ? | a sentence with enter is: as you enter you will be greeted by these rows of trees . | sent |
| ( tree ) da prayog ki ae ? | a sentence with tree is: wide shot of an eagle and its nest on top of a tree | sent |
| ( unlawful ) jidan ki hunda ae ? | the synonym for unlawful is illegal | syn |
| ( constrict ) jidan ki hunda ae ? | the synonym for constrict is close | syn |

Table 64: Sample x-y pairs in Punjabi.

| **Question** (x) | **Expected answer** (y *) | **GPT-3 generated answer** | **MemPrompt generated answer** |
|---|---|---|---|
| give me something i would hear as (whether ) ? | the homophone for whether is whither | the example of something you would hear as whether is a question word used to ask about possibilities. | the homophone for whether is weather |
| ( leper ) means what ? | the definition of leper is a person affected with leprosy. | a sentence with leper is: a leper is a person with a disease that makes his skin scaly and white. | the definition of leper is a person who has leprosy. |
| what can one confuse with ( kew ) ? | the homophone for kew is key | a word that can be confused with kew is due | the homophone for kew is cue |
| What is like ( pneumatic ) ? | the synonym for pneumatic is windy | the homophone for pneumatic is hydraulic | the synonym for pneumatic is airy |
| give me something i would hear as ( seeder ) ? | the homophone for seeder is seer | something you would hear as seeder is: someone who plants things in the ground or causes seeds to be planted. | the homophone for seeder is said |

Table 65: A subset of random samples where GPT-3-175B without memory was incorrect.

| Task and Description | Sample one iteration of FEEDBACK-REFINE |
|---|---|
| **Sentiment Reversal**<br>Rewrite reviews to reverse sentiment.<br>Dataset: [Zhang et al., 2015] 1000 review passages | $x$: The food was fantastic..."<br>$y_t$: The food was disappointing..."<br>$fb$: Increase negative sentiment<br>$y_{t+1}$: The food was utterly terrible..." |
| **Dialogue Response Generation**<br>Produce rich conversational responses.<br>Dataset: Mehri and Eskenazi [2020] 372 conv. | $x$: What's the best way to cook pasta?"<br>$y_t$: The best way to cook pasta is to..."<br>$fb$: Make response relevant, engaging, safe<br>$y_{t+1}$: Boil water, add salt, and cook pasta..." |
| **Code Optimization**<br>Enhance Python code efficiency<br>Dataset: [Madaan et al., 2023c]: 1000 programs | $x$: Nested loop for matrix product<br>$y_t$: NumPy dot product function<br>$fb$: Improve time complexity<br>$y_{t+1}$: Use NumPy's optimized matmul function |
| **Code Readability Improvement**<br>Refactor Python code for readability.<br>Dataset: [Puri et al., 2021b] 300 programs* | $x$: Unclear variable names, no comments<br>$y_t$: Descriptive names, comments<br>$fb$: Enhance variable naming; add comments<br>$y_{t+1}$: Clear variables, meaningful comments |
| **Math Reasoning**<br>Solve math reasoning problems.<br>Dataset: [Cobbe et al., 2021] 1319 questions | $x$: Olivia has $23, buys 5 bagels at $3 each"<br>$y_t$: Solution in Python<br>$fb$: Show step-by-step solution<br>$y_{t+1}$: Solution with detailed explanation |
| **Acronym Generation**<br>Generate acronyms for a given title<br>Dataset: (Appendix J.21) 250 acronyms | $x$ : Radio Detecting and Ranging"<br>$y_t$: RDR<br>$fb$ : be context relevant; easy pronunciation<br>$y_{t+1}$: RADAR" |
| **Constrained Generation**<br>Generate sentences with given keywords.<br>Dataset: [Lin et al., 2020a] 200 samples | $x$: beach, vacation, relaxation<br>$y_t$: During our beach vacation...<br>$fb$: Include keywords; maintain coherence<br>$y_{t+1}$: .. beach vacation was filled with relaxation |

Table 66: An overview of the tasks which we evaluate SELF-REFINE on, along with their associated datasets and sizes. For every task, we demonstrate a single iteration of refinement of input $x$, the previously generated output $y_t$, the feedback generated $fb_t$, and the refinement $y_{t+1}$. Few-shot prompts used for FEEDBACK and REFINE are provided in Appendix J.23.

## J.2 Broader Related Work

Compared to a concurrent work, Reflexion Shinn et al. [2023], our approach involves correction using feedback, whereas their setup involves finding the next best solution in planning using ReAct. While ReAct and Reflexion provide a free-form reflection on whether a step was executed correctly and potential improvements, our approach is more granular and structured, with multi-dimensional feedback and scores. This distinction allows our method to offer more precise and actionable feedback, making it suitable for a wider range of natural language generation tasks, including those that may not necessarily involve step-by-step planning such as open-ended dialogue generation.

**Comparison with Welleck et al. [2022]** The closest work to ours may be Self-Correction [Welleck et al., 2022]; however, Self-Correction has several disadvantages compared to SELF-REFINE:

1. Self-Correction does not train their model to generate explicit feedback; instead, Welleck et al. [2022] trained their models to refine only. As we show in Section 10.4 and Table 10.2, having the model generate explicit feedback results in significantly better refined outputs.
2. Self-Correction trains a separate refiner (or "corrector") for each task. In contrast, SELF-REFINE uses instructions and few-shot prompting, and thus does not require training a separate refiner for each task.
3. Empirically, we evaluated SELF-REFINE using the same base model of GPT-3 as Self-Correction, and with the same settings on the GSM8K benchmark. Self-Correction achieved 45.9% accuracy while SELF-REFINE (this work) achieved **55.7**% (↑**9.8**).

**Comparison with non-refinement reinforcement learning (RL) approaches.** Rather than having an explicit refinement module, an alternative way to incorporate feedback is by optimizing a scalar reward function, e.g. with reinforcement learning (e.g., Stiennon et al. [2020], Lu et al. [2022], Le et al. [2022a]). These methods differ from SELF-REFINE (and more generally, refinement-based approaches) in that the model cannot access feedback on an intermediate generation. Second, these reinforcement learning methods require updating the model's parameters, unlike SELF-REFINE.

## J.3 Human Evaluation

The A/B evaluation in our study was conducted by the authors, where a human judge was presented with an input, task instruction, and two candidate outputs generated by the baseline method and SELF-REFINE. The setup was blind, i.e., the judges did not know which outputs were generated by which method. The judge was then asked to select the output that is better aligned with the task instruction. For tasks that involve A/B evaluation, we calculate the relative improvement as the percentage increase in preference rate. The preference rate represents the proportion of times annotators selected the output produced by SELF-REFINE over the output from the baseline method. Table 67 shows the results.

While multiple annotators participated in each task, we only collected a single annotation per instance, aiming to scale the number of data points we could annotate. To further validate the reliability of our evaluations, we obtained two annotations for 50 samples from each dataset. All human evaluations were executed in a double-blind manner, with the responses being randomly flipped to ensure annotator impartiality, leaving them unaware of whether a given output was from the base model or the SELF-REFINE. These additional evaluations were exclusively applied to outputs generated by GPT-4.

For each task, we measured inter-labeler agreement using Cohen's Kappa score. Code Readability Improvement and Acronym Generation both scored a substantial 0.75, Sentiment Reversal was also substantial at 0.61, while Dialogue Response Generation was moderate with a score of 0.53.

| **Task** | **SELF-REFINE (%)** | **Direct (%)** | **Either (%)** |
|---|---|---|---|
| Sentiment Reversal | 75.00 | 21.43 | 3.57 |
| Acronym Generation | 44.59 | 12.16 | 43.24 |
| Dialogue Response Generation | 47.58 | 19.66 | 32.76 |
| Code Readability Improvement | **50.0** | 3.0 | 47.0 |

Table 67: Relative improvement of SELF-REFINE in A/B evaluations across different tasks. The values represent normalized preferences, which correspond to the proportion of times the output generated by SELF-REFINE was selected as better aligned with the task instruction over the baseline method. The evaluation was conducted for 150 examples for each dataset. The judges were not aware of the method that generated each sample.

## J.4 SELF-REFINE on Reasoning Tasks

SELF-REFINE shows limited success in Math Reasoning tasks, mainly due to its challenges in generating meaningful feedback. Nonetheless, we observe performance gains with SELF-REFINE in certain tasks from the Big-Bench Hard suite [Suzgun et al., 2022b], particularly those requiring commonsense reasoning and logical thinking. These results in Table 68 suggest that SELF-REFINE is more effective in scenarios where the combination of logical reasoning and pre-trained knowledge contribute to self-verification.

| Task | Base model | +Self-Refine | Gain |
|---|---|---|---|
| Date Understanding | 62.0 | 66.8 | 4.8 |
| Geometric Shapes | 17.6 | 20.0 | 2.4 |
| Logical Deduction (seven objects) | 43.2 | 45.2 | 2.0 |
| Multi-Step Arithmetic | 61.6 | 64.0 | 2.4 |
| Tracking Shuffled Objects (seven objects) | 31.6 | 36.0 | 4.4 |

Table 68: Performance comparison between the base outputs and the SELF-REFINE enhanced model across various Big Bench Hard tasks. All experiments were conducted using the GPT-3.5-TURBO-0613 model with a temperature setting of 0.0. No task-specific prompts were utilized; all tasks employed the same instructional prompts.

## J.5 Instruction-Only Prompting

In our main experiments, we use few-shot prompting to guide model output into a more readily parseable format. Next, we experiment with SELF-REFINE under a zero-shot prompting scenario, where traditional few-shot examples are supplanted by explicit instructions at each stage of the SELF-REFINE process. For these experiments, we use GPT-3.5. The results in Table 69) show that SELF-REFINE remains effective across diverse tasks. However, achieving optimal performance in this setting requires extensive prompt engineering for instructions. The instructions are present in Listing 1 (Acronym Generation), Listing 3 (Math Reasoning), Listing 2 (Sentiment Reversal), Listing 4 (Constrained Generation), and Listing 5 (Dialogue Response Generation).

For Math Reasoning tasks, SELF-REFINE improves the solve rate from 22.1% to 59.0% in an instruction-only setting. We find that much of this gain comes from fixing omitted return statements in 71% of the initial Python programs, despite clear instructions to include them. Subsequent iterations of feedback generation and refinement address this issue effectively, decreasing the error rate by 19%. Further, we find that when the initial programs are valid, SELF-REFINE does not improve the performance.

## J.6 GPT-4 Evaluation

In light of the impressive achievements of GPT-4 in assessing and providing reasoning for complex tasks, we leverage its abilities for evaluation in SELF-REFINE. The approach involves presenting tasks to GPT-4 in a structured manner, promoting the model's deliberation on the task

| Task | Base | SELF-REFINE (zero-shot) |
|---|---|---|
| Acronym Generation | 16.6% | 44.8% (↑ 28.2%) |
| Constrained Generation | 4.0% | 47.0% (↑ 43.0%) |
| Sentiment Reversal | 4.4% | 71.4% (↑ 67.0%) |
| Math Reasoning | 22.1% | 59.0% (↑ 36.9%) |
| Dialogue Response Generation | 23.0% | 48.8% (↑ 25.8%) |

Table 69: Performance of SELF-REFINE with Zero-Shot Prompting

**Listing 1** Instruction-only prompts used at various stages of the SELF-REFINE process for Acronym Generation.

```
# Init: Generate an acronym for a given title
start_chat_log = [
    {"role": "system", "content": 'You are a helpful assistant that
    ↪  generates acronyms.'},
    {"role": "user", "content": f'Generate an acronym for the title
    ↪  "{title}". Please respond in the format: "The generated acronym is:
    ↪  {acronym}"'}
]

# Generate Feedback: Evaluate the quality of the generated acronym
feedback_str = f"""Evaluate the acronym "{acronym}" for the title "{title}"
↪  on its ease of pronunciation, ease of spelling, relation to the title,
↪  positive connotation, and being well-known."""
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that provides
    ↪  feedback on the viability and quality of acronyms.'},
    {"role": "user", "content": feedback_str}
]

# Refine: Improve the acronym based on provided feedback
start_chat_log = [
    {"role": "system", "content": 'You are a helpful assistant that can
    ↪  iteratively improve acronyms based on feedback.'},
    {"role": "user", "content": f'Improve the acronym "{acronym}" for the
    ↪  title "{title}" based on the following feedback: {feedback}. Please
    ↪  *always* respond in the format: "The improved acronym is:
    ↪  {acronym}"'}
]
```

and generating a rationale for its decision. To mitigate order bias in our tasks, we randomly flip the order of the outputs generated after the first iteration and by SELF-REFINE before evaluation. Further, to ensure that there is no inherent bias in GPT-4 picking its own predictions, we conduct an additional study with CLAUDE-2 .

**Listing 2** Instruction-only prompts used at various stages of the Self-Refine process for Sentiment Reversal.

```
# Init: Transform a negative sentiment review into a positive one
start_chat_log = [
    {"role": "system", "content": 'You are an AI language model that
    ↪  transforms a negative sentiment review into a positive one.'},
    {"role": "user", "content": f'Please transform the following negative
    ↪  review into a positive one: "{review}". Respond in the format of:
    ↪  "The positive review is: [Your Response Here]".'},
]

# Generate Feedback: Provide feedback on the transformed review
start_chat_log = [
    {"role": "system", "content": 'You are an AI model providing feedback
    ↪  on a sentiment transformed review.'},
    {"role": "user", "content": f"Why is this review not Very positive?
    ↪  Point out specific issues and give concrete suggestions. Respond in
    ↪  the format of: 'Feedback: [Specific issues and suggestions]"},
]

# Refine: Improve the review based on provided feedback
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that improves upon
    ↪  an existing review based on provided feedback.'},
    {"role": "user", "content": f'Please improve the sentiment of the
    ↪  following review "{sentence}" based on this feedback: "{feedback}",
    ↪  and make it more positive. Always respond in the format of: "The
    ↪  more positive review is: [Your Response Here]".'},
]
```

**Claude-2 as the Evaluator** Despite the measures we took to prevent any inherent biases, GPT-4 might inherently favor self-refined outputs. To provide a comprehensive evaluation of GPT-4, we conduct an additional analysis using CLAUDE-2, [16] serving as an independent evaluator for GPT-4 outputs. The results in Table 70 from GPT-4 as the base LLM with CLAUDE-2 as the evaluator show the same strong preferences for SELF-REFINE over the Base outputs.

This prompts used for evaluation are listed in Listings 6 to 8.

## J.7 Model Key

We use terminology here: https://platform.openai.com/docs/models/gpt-3-5. The experiments were done with the 0313 versions of GPT-4 and GPT-3.5 unless otherwise mentioned. We refer to text-davinci-003 as GPT-3. We use GPT-3.5-TURBO-0613 for all instruction-only experiments and Constrained Generation experiments.

[16] https://www.anthropic.com/index/claude-2

**Listing 3** Instruction-only prompts used at various stages of the Self-Refine process for Math Reasoning.

```
# Function to Generate an Answer
gen_sys_template = "You are a helpful assistant that responds with only a
 ↪  python program."
gen_user_template = """Write a python function that solves the given
 ↪  question and returns the result.
Always store the final result in a variable called `result`, and always
 ↪  include `return result` as your last statement.
# Question: {question}
# solution using Python:"""

# Function to Get Feedback on the Answer
fb_sys_template = "You are a helpful assistant that provides feedback on
 ↪  the correctness of python programs."
fb_user_template = """Question: {question}
{prediction}
# There may be an error in the code above because of lack of understanding
 ↪  of the question.
To find the error, go through semantically complete blocks of the code, and
 ↪  check if everything looks good.
Errors could include:
- Logical issues
- Lack of understanding of the question
- Missing `return result`
Share your evaluation in the format:
Evaluation: <your evaluation here>.
If everything looks good, return 'Evaluation: correct'"""

# Function to Fix the Error
refine_sys_template = "You are a helpful assistant that responds with only
 ↪  a python program."
refine_user_template = """Question: {question}
{prediction}
# There is an error in the code above. The following is the feedback on the
 ↪  code:
{feedback}
Fix the error in the python function given the feedback above. The python
 ↪  function should return the final answer."""
```

| Task | % Base | % SELF-REFINE |
|---|---|---|
| Dialogue Response Generation | 30.6 | **64.7** (↑34.1) |
| Sentiment Reversal | 10.6 | **69.2** (↑58.6) |
| Acronym Generation | 32.0 | **49.2** (↑17.2) |
| Code readability | 37.0 | **60.0** (↑23.0) |

Table 70: Evaluation Results of GPT-4 with CLAUDE-2 as Evaluator

**Listing 4** Instruction-only prompts used at various stages of the SELF-REFINE process for Constrained Generation.

```
# Constrained Generation Task

# 1. Init
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that generates
    ↪  sentences with commonsense, using a given set of concepts.'},
    {"role": "user", "content": f'Generate a commonsense sentence using the
    ↪  following concepts: {", ".join(concepts)}. Please respond in the
    ↪  format: "The generated sentence is: {sentence}"'},
]

# 2. Get Feedback
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that provides
    ↪  feedback on a sentence generated with specific concepts.'},
    {"role": "user", "content": f'''Evaluate the following sentence
    ↪  "{sentence}", which was meant to use the following concepts: {",
    ↪  ".join(concepts)}.
    Please provide two pieces of feedback. Start your feedback about
↪   concept usage with the phrase "Concept feedback is:", and start your
↪   feedback about commonsense facts with the phrase "Commonsense feedback
↪   is:".
    The format should be:
    Concept feedback is: <list of missing concepts>
    Commonsense feedback is: <commonsense feedback here>'''},
]

# 3. Iterate Fix
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that improves upon
    ↪  an existing sentence based on provided feedback.'},
    {"role": "user", "content": f'Improve upon the following sentence
    ↪  "{improved_sentence}" based on the following feedback:
    ↪  {feedback_string}. Please respond in the format: "The improved
    ↪  sentence is: [Your improved sentence here]"'},
]
```

**Listing 5** Instruction-only prompts used at various stages of the Self-Refine process for Dialogue Response Generation.

```
# Dialogue Response Generation Task

# 1. Generate Response
start_chat_log = [
    {"role": "system", "content": 'You are a helpful assistant that
    ↪  generates responses.'},
    {"role": "user", "content": f'Provided a dialogue between two speakers,
    ↪  generate a response that is coherent with the dialogue history.
    ↪  Desired traits for responses are: 1) Relevant - The response
    ↪  addresses the context, 2) Informative - The response provides some
    ↪  information, 3) Interesting - The response is interesting, 4)
    ↪  Consistent - The response is consistent with the rest of the
    ↪  conversation in terms of tone and topic, 5) Helpful - The response
    ↪  is helpful in providing any information or suggesting any actions,
    ↪  6) Engaging - The response is engaging and encourages further
    ↪  conversation, 7) Specific - The response contains specific content,
    ↪  8) Safe - The response should not cause danger, risk, or injury 9)
    ↪  User understanding - The response demonstrates an understanding of
    ↪  the user\'s input and state of mind, and 10) Fluent. Response
    ↪  should begin with - Response:\n\nConversation
    ↪  history:\n\n{history}\n\nResponse: '},
]

# 2. Get Feedback
start_chat_log = [
    {"role": "system", "content": 'You are an AI model that provides
    ↪  feedback on the viability and quality of responses.'},
    {"role": "user", "content": feedback_str}
]

# 3. Iterate Response
start_chat_log = [
    {"role": "system", "content": 'You are a helpful assistant that can
    ↪  iteratively improve responses based on feedback.'},
    {"role": "user", "content": iterate_str}
]
```

**Listing 6** Prompt for GPT-4 evaluation of Sentiment Reversal.

```
f"""Which review is aligned with the sentiment {target_sentiment}?
Review A: {review_a}
Review B: {review_b}.

Pick your answer from ['Review A', 'Review B', 'both', 'neither']. Generate
 ↪  a short explanation for your choice first. Then, generate 'The more
 ↪  aligned review is A' or 'The more aligned review is B' or 'The more
 ↪  aligned review is both' or 'The more aligned review is neither'.

Format: <explanation> <answer> STOP
```

**Listing 7** Prompt for GPT-4 evaluation of Acronym Generation.

```
f"""Title: {title}

Acronym A: {acronym_a}
Acronym B: {acronym_b}

Pick the better acronym for the given title. The acronyms should be
 ↪  compared based on the following criteria:
* Ease of pronunciation.
* Ease of spelling.
* Relation to title.
* Positive connotation.

Generate your answer in the following format:

<Short explanation>. The better acronym is A OR The better acronym is B OR
 ↪  The acronyms are equally good OR Neither acronym is good. STOP.
```

**Listing 8** Prompt for GPT-4 evaluation of Dialogue Response Generation.

```
f"""Which response is better given this context: {context}?
Response A: {response_a}


Response B: {response_b}.

Pick your answer from ['Response A', 'Response B', 'both', 'neither'].
 ↪  Generate a short explanation for your choice first. Then, generate 'The
 ↪  better response is A' or 'The better response is B' or 'The better
 ↪  response is both' or 'The better response is neither'.

Format: <explanation> <answer> STOP
```

**Listing 9** Prompt for GPT-4 evaluation of Constrained Generation

```
f"""Which story is better?
Story A: {story_a}

Story B: {story_b}.

Judge the story based on the flow, the grammar, and the overall quality of
 ↪  the story. Rate more realistic story higher. Pick your answer from
 ↪  ['Story A', 'Story B', 'either']. First, reason about your choice. Then,
 ↪  generate 'The better story is Story A' or 'The better story is Story B'
 ↪  or 'The better story is either'.

Format:

Reasoning: <your reasoning>. The better story is <your choice>.


Reasoning:"""
```

## J.8 Comparison of SELF-REFINE with State-of-the-art of Few-Shot Learning Models and Fine-Tuned Baselines

In this section, we present a comprehensive comparison of the performance of SELF-REFINE with other few-shot models and fine-tuned baselines across a range of tasks, including mathematical reasoning and programming tasks. Tables 71 and 72 display the performance of these models on the GSM tasks and PIE dataset, respectively. Table 73 shows the performance of the CODEX model with and without SELF-REFINE on the Code Readability task, further described in Appendix J.16. Our analysis demonstrates the effectiveness of different model architectures and training techniques in tackling complex problems.

| **Method** | | **Solve Rate** |
|---|---|---|
| Chen et al. [2021c] | CODEX | 71.3 |
| Cobbe et al. [2021] | OpenAI 6B | 20.0 |
| Wei et al. [2022b] | CoT w/ CODEX | 65.6 |
| Gao et al. [2023] | PaL w/ CODEX | 72.0 |
| | PaL w/ GPT-3 | 52.0 |
| | PaL w/ GPT-3 | 56.8 |
| | PaL w/ GPT-3.5 | 74.2 |
| | PaL w/ GPT-4 | 93.3 |
| Welleck et al. [2022] | Self-Correct w/ GPT-3 | 45.9 |
| | Self-Correct (fine-tuned) | 24.3 |
| This work | SELF-REFINE w/ GPT-3 | **55.7** |
| | SELF-REFINE w/ GPT-3 | **62.4** |
| | SELF-REFINE w/ GPT-3.5 | **75.1** |
| | SELF-REFINE w/ GPT-4 | **94.5** |
| | SELF-REFINE w/ CODEX (Oracle Feedback) | **76.2** |

Table 71: Performance comparison of models on math reasoning (Math Reasoning).

| Method | | %OPT |
|---|---|---|
| Puri et al. [2021b] | **Human References** | 38.2 |
| OpenAI Models: OpenAI [2022, 2023] | CODEX | 9.7 |
| | GPT-3 | 14.8 |
| | GPT-3.5 | 22.2 |
| | GPT-4 | 27.3 |
| Nijkamp et al. [2022b] | CODEGEN-16B | 1.1 |
| Berger et al. [2022] | SCALENE | 1.4 |
| | SCALENE (BEST@16) | 12.6 |
| | SCALENE (BEST@32) | 19.6 |
| Madaan et al. [2023c] | -2B | 4.4 |
| | -2B (BEST@16) | 21.1 |
| | -2B (BEST@32) | 26.3 |
| | -16B | 4.4 |
| | -16B (BEST@16) | 22.4 |
| | -16B (BEST@32) | 26.6 |
| | -Few-shot (BEST@16) | 35.2 |
| | -Few-shot (BEST@32) | **38.3** |
| This work | SELF-REFINE w/ CODEX | 15.6 |
| | SELF-REFINE w/ GPT-3 | 23.0 |
| | SELF-REFINE w/ GPT-3.5 | 26.7 |
| | SELF-REFINE w/ GPT-4 | 36.0 |

Table 72: Performance comparison of various models on the PIE dataset in terms of the percentage of programs optimized (%OPT). The table includes human references, baseline models, fine-tuned -2B and -16B models, and our proposed model (SELF-REFINE) using different LLMs. Notably, SELF-REFINE achieves superior performance while using only 4 samples at most, significantly fewer than the 16 and 32 samples employed by other models. Scalene, an off-the-shelf optimizer, uses instruction tuning with CODEX and serves as a comparison point.

| Method | | %Readable Variables |
|---|---|---|
| Chen et al. [2021c] | CODEX | 37.4 |
| This work | SELF-REFINE w/ CODEX | 51.3 |

Table 73: Performance of SELF-REFINE on CODEX on the Code Readability task. Further details on the task are described in Appendix J.16

## J.9 Evaluation of Vicuna-13b

We also experiment with Vicuna-13b [Chiang et al., 2023], a version of LLaMA-13b [Touvron et al., 2023] fine-tuned on conversations sourced from the web. Vicuna-13b was able to consistently follow the task initialization prompt. However, it struggled to follow the prompts intended for feedback and refinement. This often led to outputs that resembled assistant-like responses.

It's important to note that we used the same prompts for Vicuna-13b as those used with other models in our study. However, the limited performance of Vicuna-13b suggests that this model may require more extensive prompt-engineering for optimal performance.

**Mixed-refine: Improving Vicuna-13b with GPT-3.5** While the focus of SELF-REFINE is improvement of the model without any external help, it may be possible to use a smaller model for the initialization, and then involving a bigger model for refinement. To test this, we experiment with a setup where we use Vicuna-13b as the initialization model, and use GPT-3.5 as the FEEDBACK and REFINE. The results on Math Reasoning show the promise of this approach: while Vicuna-13b was able to get only 24.18% on Math Reasoning, it was able to improve to 40.5% in this mixed-refinement setting.

## J.10 SELF-REFINE wiht LLAMA-2-70B

We further benchmark SELF-REFINE, this time on the open-access model Lllama-2 Touvron et al. [2023] in an instruction-only setting (no few-shot prompts, only instructions). The results in Table 74 show that SELF-REFINE continues to be effective on an open-access model and without few-shot examples at all. Given these performance metrics, alongside anticipated advancements in hardware, we anticipate the broad and cost-effective applicability of SELF-REFINE.

| Task | Base | Self-Refine | Equally Good |
|---|---|---|---|
| Acronym Generation | 22.30 | 53.08 | 22.30 |
| Sentiment Reversal | 13.2 | 60.8 | 26 |
| Dialogue Response Generation | 11.2 | 20.4 | 54.6 |
| Math Reasoning | 37.6 | 37.8 (41 with Oracle) | N/A |

Table 74: Instruction-only (zero-shot) results using SELF-REFINE on the open-access model Llama-2. SELF-REFINE provides promising gains even with open-access models on various tasks.

## J.11 Self-Refining Images

We experiment with applying SELF-REFINE for iterative refinement of images using a multimodal variant of GPT-4, GPT4-V [17]. We start the process (INIT) by generating TikZ [Tantau, 2013] code for an object, such as an animal or an illustrative diagram. The Tikz code is compiled into an image. The generated image is added to the feedback prompt of GPT4-V for feedback generation. This feedback is then again used by GPT4-V's to rewrite and refine the original TikZ code, thereby improving the visual output in each cycle. A self-contained Colab notebook for visual SELF-REFINE available at https://github.com/madaan/self-refine/blob/main/colabs/Visual-Self-Refine-GPT4V.ipynb.

[17] https://openai.com/research/gpt-4v-system-card

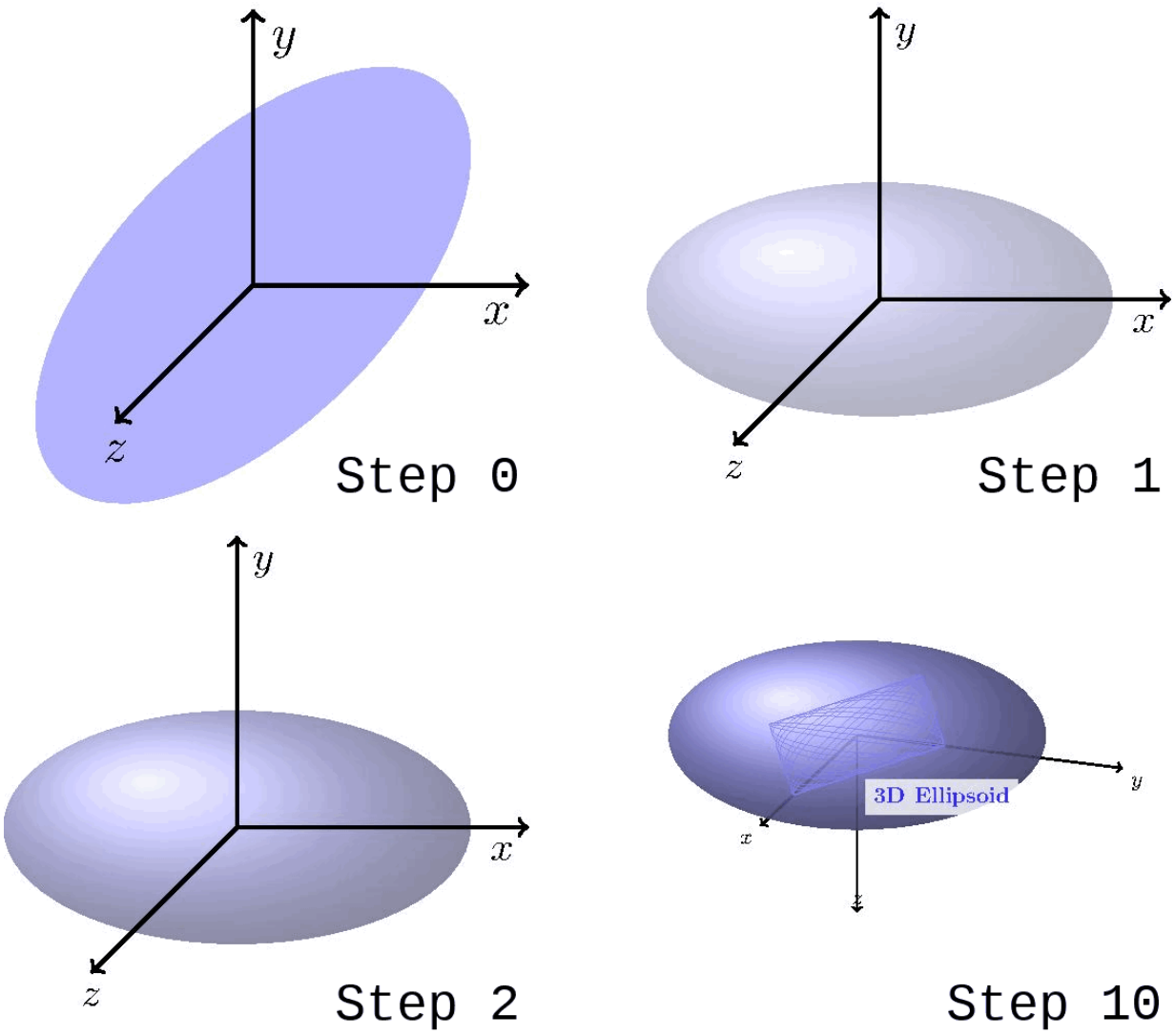



Figure 66: Visual SELF-REFINE: Refining a diagram of 3d Ellipsoid.More examples at https://github.com/madaan/self-refine/tree/main/docs/visual_self_refine_examples for more examples.

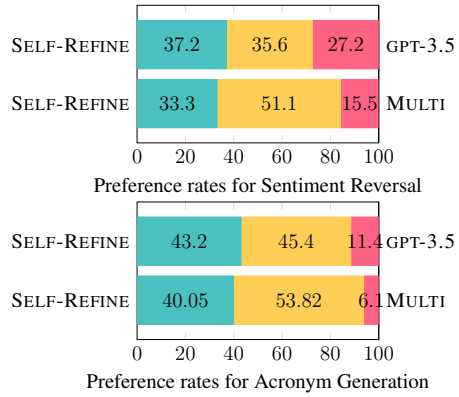



Figure 67: Preference for the outputs generated by our method (SELF-REFINE), the multiple-sample baseline (MULTI), and ties (ties).

| | GPT-3 | | GPT-3.5 | | GPT-4 | |
|---|---|---|---|---|---|---|
| Task | Base | +SELF-REFINE | Base | +SELF-REFINE | Base | +SELF-REFINE |
| Math Reasoning | **64.1** | **64.1** (0) | 74.8 | **75.0** (↑0.2) | 92.9 | **93.1** (↑0.2) |
| Math Reasoning (Oracle) | 64.06 | **68.9** (↑4.8) | 74.8 | **76.2** (↑1.4) | 92.9 | **93.8** (↑0.7) |

Table 75: SELF-REFINE results on Math Reasoning using GPT-3, GPT-3.5, and GPT-4 as base LLM with Oracle feedback.

## J.12 Additional Analysis

### Using Oracle Feedback

We experimented with *Oracle Feedback* following Welleck et al. [2022]. This method uses correctness information to guide model refinement, only progressing to REFINE stage if the current answer is incorrect. This adjustment notably enhanced performance in the Math Reasoning task, with GPT-3 improving by 4.8% and GPT-4 by 0.7% Table 75. This indicates the potential of external signals to optimize model performance in particular tasks.

| Iteration | Acronym | Pronunciation | Pron. (5) | Spell. (5) | Rel. (5) | Pos. Con. (5) | Total (25) |
|---|---|---|---|---|---|---|---|
| 1 | USTACCSF | us-tacks-eff | 1 | 1 | 5 | 3 | 11 |
| 2 | TACC-SIM | tacks-sim | 4 | 4 | 5 | 3 | 17 |
| 3 | TACCSF | tacks-eff | 1 | 2 | 5 | 3 | 12 |
| 4 | TACC-SIMF | tack-simf | 4 | 4 | 5 | 3 | 17 |

Table 76: Acronym generation results across iterations, showcasing how improvements in certain aspects (e.g., pronunciation and spelling) can be accompanied by losses in others, leading to fluctuating overall performance in multi-aspect feedback tasks like Acronym Generation.

| Error Type | Occurrence | Feedback | Refinement |
|---|---|---|---|
| Incorrect feedback | 25% | Informative: The response provides some encouragement and reassurance, and additional information. 3/3 | Response: That's hilarious! Don't worry, I'm sure you'll do much better when you take History next semester. Are you looking forward to it? |
| Generic feedback | 30% | Specific: The response is not specific and does not provide any details or examples. 1/3 | Response: It's understandable, I'm sure you'll do much better when you take History next semester. Maybe you can use your knowledge of Christopher Columbus to ace your next test! Are you looking forward to it? |
| Incorrect Scoring | 10% | Informative: The response provides a supportive and understanding message, which is informative to the user. 2/3 | Response: I'm sorry to hear... when Christopher Columbus didn't even fight in the Civil War! I'm sure you'll do much better.... Are you looking forward to it? |

Table 77: Error analysis for Dialogue Response Generation: When the feedback is not useful, a large majority is not specific or incorrect.

**Non-monotonic increase in output quality for acronym generation** For tasks with multi-aspect feedback like Acronym Generation, the output quality can fluctuate during the iterative process, improving on one aspect while losing out on another (Table 76). To address this, SELF-REFINE's feedback generates explicit numerical scores to capture the different aspects of output quality. This allows for a more balanced evaluation of outputs and the selection of the most appropriate one. The algorithm selects the best output based on the maximum score across all iterations, as described in Algorithm 2 (line 8). A similar selection is possible for other tasks like Math Reasoning and Sentiment Reversal, while we observe that output quality increases monotonically with iterations.

**Feedback and Refinement Error Analysis for Response Generation** We perform a detailed error analysis of SELF-REFINE feedback and refinement process for Dialogue Response Generation, which we summarize in Tables Table 77 and Table 78.

Table 77 reports the occurrence of different types of errors in our sample, which includes Incorrect Feedback (25%), Generic Feedback (30%), and Incorrect Scoring (10%). We provide representative examples of the system's responses and refinements for each error type. These errors highlight potential areas for improving our feedback handling mechanism, particularly in the interpretation and understanding of user inputs.

Table 78 breaks down errors found in the refinement stage of SELF-REFINE. Errors include: not being robust (10%), ignoring feedback (25%), and introducing a new problem (20%). We demonstrate how the model handles a variety of feedback types, how robust it is under different

| Error Type | Occurrence | Feedback | Refinement |
|---|---|---|---|
| Not-Robust | 10% | Specific: The response is specific and provides specific details or examples. 3/3 | Response: That's pretty funny! It's understandable that you didn't do well on the test. I'm sure you'll do much better when you take History next semester |
| Ignores feedback | 25% | Informative: The response doesn't provide any new information. 1/3 | Response: That's so kind and generous of you! I would love to hear more about the podcasts you enjoy. Do you have any favorites you can recommend? |
| Introduces new problem | 20% | Informative: 3/3 | Response: That would be a disaster! Luckily, my coworkers know better than to drink while they work. New Informative score: 2/3 |
| Robust to bad feedback | 60% | Informative: The response does not provide any information. 1/3 | Response: Wow, 25 people! That must have been an amazing experience. Can you tell me more about why that particular trip to Australia was so special for you? |

Table 78: On the Dialogue Response Generation task, SELF-REFINE can ignore good feedback but in a majority of cases, it is robust to bad feedback and ignores bad feedback.

circumstances, and how often it inadvertently introduces new issues. 60% of the times, the model is robust to incorrect or generic feedback. These insights can guide us in enhancing the model's refinement capabilities, especially in providing accurate and specific responses.

## J.13 Beyond Benchmarks

SELF-REFINE demonstrates its iterative feedback and refinement capabilities in the context of website layout generation. GPT-3.5 initially produces a rudimentary layout for a given topic, and then uses the FEEDBACK to suggest specific, actionable improvements, as demonstrated in Figures 68 and 70. These suggestions range from design changes such as color and font adjustments, to content enhancements and layout modifications. Figures 69 and 71 showcase the final layouts, post-feedback implementation, highlighting the potential and versatility of SELF-REFINE across different scenarios.

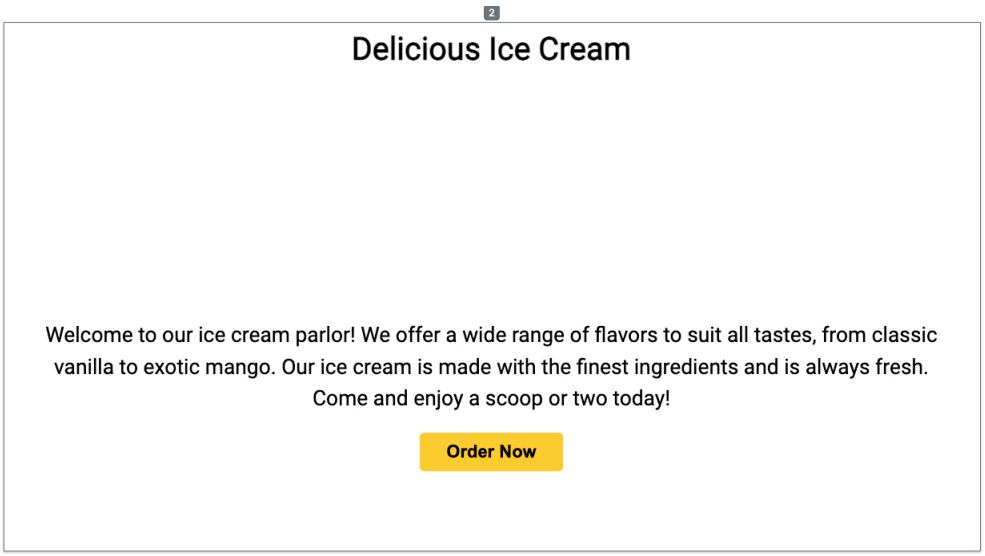



Figure 68: Initial web layout generated by our model for a fictional ice cream parlor.

**Ice Cream Generation** The feedback generated by FEEDBACK for ice cream generation:

- Change the background color of the container to a light blue color (#6f2ff).
- Change the font size of the heading to 48px.
- Add a small icon before the "Welcome to our ice cream parlor!" text using the URL https://cdn-icons-png.flaticon.com/512/3622/3622340.png.
- Add an additional paragraph after the existing text with the following text: "We also offer a variety of toppings and cones to complement your ice cream. Visit us today to try our latest flavors and indulge in a sweet treat!"
- Increase the font size of the button text to 24px.
- Update the button color to #9933.

**Photosynthesis** The feedback generated by FEEDBACK for photosynthesis:

- Increase the font size of the text to 18px for better readability.
- Add more information about the benefits of photosynthesis.

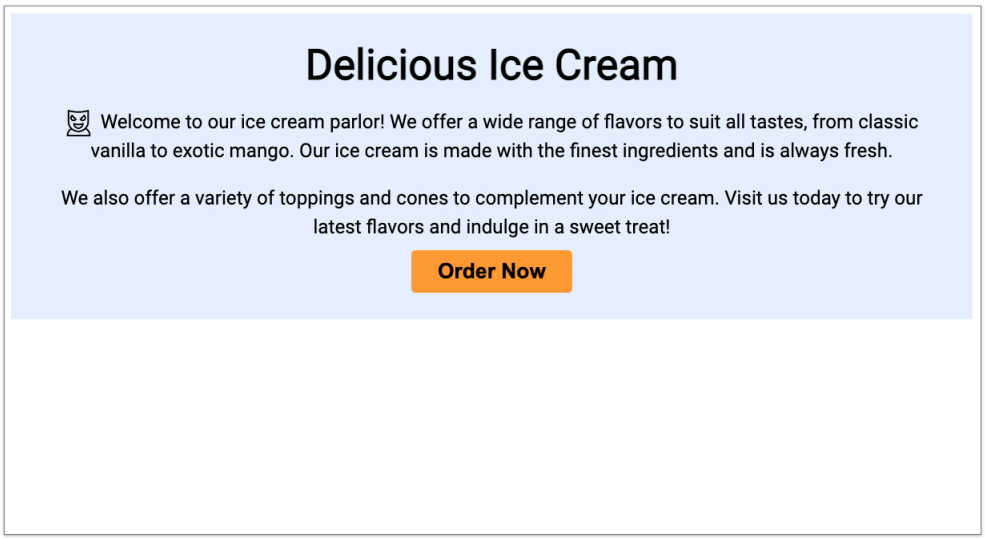



Figure 69: Refined web layout after applying model feedback. The feedback included changing the background color to light blue (#6f2ff), increasing the heading font size to 48px, adding an icon before the welcome text, enhancing the content with an additional paragraph, increasing the button text size to 24px, and updating the button color to #9933.

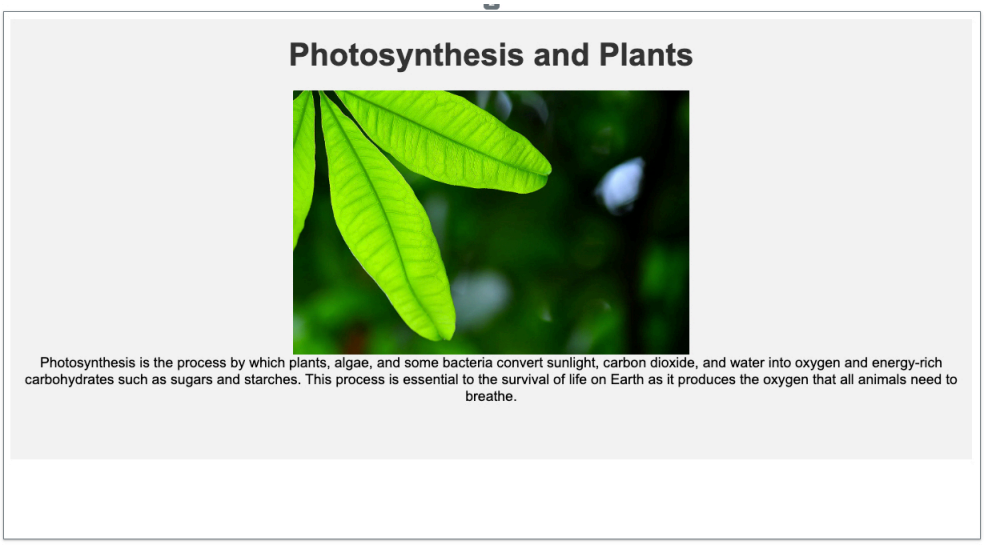



Figure 70: Initial web layout generated by our model for a page on photosynthesis.

- Remove the unnecessary margin-top from the header.
- Add a ruler or divider below the header to separate it from the image.

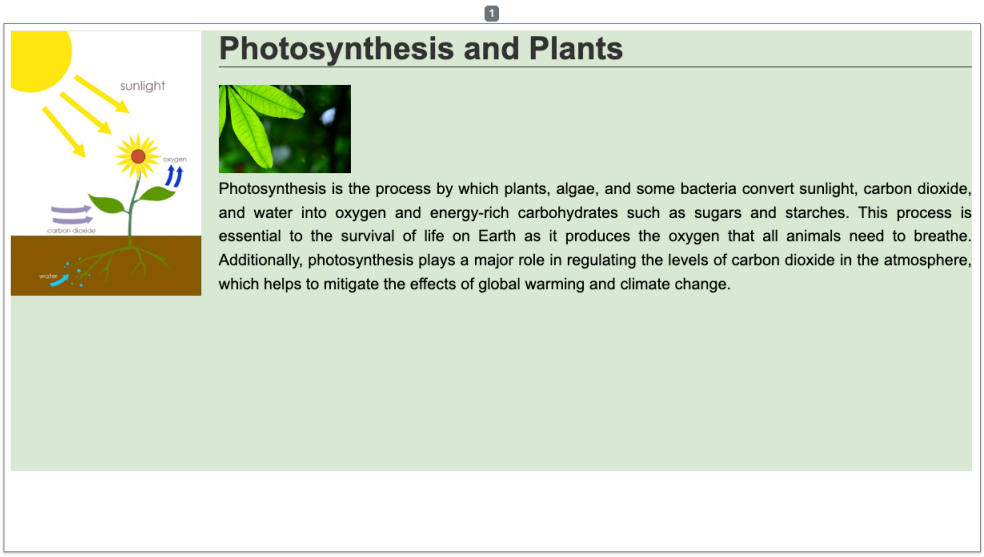



Figure 71: Refined web layout after applying model feedback. The feedback included increasing the text font size to 18px for better readability, adding more information about the benefits of photosynthesis, removing the unnecessary margin-top from the header, and adding a ruler or divider below the header to separate it from the image.

## J.14 Statistical Confidence Intervals

| | GPT-3 | | GPT-3.5 | | GPT-4 | |
|---|---|---|---|---|---|---|
| Task | Base | SELF-REFINE | Base | SELF-REFINE | Base | SELF-REFINE |
| Sentiment Reversal | 8.8 ± 2.0 | **30.4** ± 3.6* | 11.4 ± 2.3 | **43.2** ± 4.0* | 3.8 ± 1.3 | **36.2** ± 3.8* |
| Dialogue Response | 36.4 ± 6.1 | **63.6** ± 6.6* | 40.1 ± 6.3 | **59.9** ± 6.7* | 25.4 ± 5.4 | **74.6** ± 6.2* |
| Code Optimization | 14.8 ± 2.7 | **23.0** ± 3.2* | 23.9 ± 3.3 | **27.5** ± 3.5 | 27.3 ± 3.5 | **36.0** ± 3.8* |
| Code Readability | 37.4 ± 6.86 | **51.3** ± 7.4 | 27.7 ± 6.1 | **63.1** ± 7.4* | 27.4 ± 6.1 | **56.2** ± 7.4* |
| Math Reasoning | **64.1** ± 3.5 | **64.1** ± 3.5 | 74.8 ± 3.2 | **75.0** ± 3.2 | 92.9 ± 2.0 | **93.1** ± 2.0 |
| Acronym Gen. | 41.6 ± 7.7 | **56.4** ± 8.1 | 27.2 ± 6.6 | **37.2** ± 7.5 | 30.4 ± 6.9 | **56.0** ± 8.1* |
| Constrained Gen. | 28.0 ± 7.38 | **37.0** ± 8.3 | 44.0 ± 8.7 | **67.0** ± 9.0* | 15.0 ± 5.4 | **45.0** ± 8.8* |

Table 79: SELF-REFINE results from table 10.1 with Wilson confidence interval (at 95% confidence interval) and statistical significance. On various tasks using GPT-3, GPT-3.5, and GPT-4 as base LLM, SELF-REFINE consistently improves LLM. Metrics used for these tasks are defined in Section 10.3.2 as follows: Math Reasoning uses the solve rate; Code Optimization uses the percentage of programs optimized; and Sentiment Reversal, Dialogue Response and Acronym Gen use a GPT-4-based preference evaluation, which measures the percentage of times outputs from the base or enhanced models were selected, with the rest categorized as a tie. Constrained Gen uses the coverage percentage. Gains over Base, that are statistically significant based on these confidence intervals are marked *

Table 79 shows results from Table 10.1 with Wilson confidence interval Brown et al. [2001] (at $\alpha$= 99% confidence interval) and statistical significance. Gains that are statistical significance based on these confidence intervals are marked with an asterisk. We find that nearly all of GPT-4 gains are statistically significant, GPT-3.5 gains are significant for 4 out of 7 datasets, and GPT-3 gains are significant for 3 out of 7 datasets.

## J.15 New Tasks

**Constrained Generation** We introduce "CommonGen-Hard," a more challenging extension of the CommonGen dataset Lin et al. [2020a], designed to test state-of-the-art language models' advanced commonsense reasoning, contextual understanding, and creative problem-solving. CommonGen-Hard requires models to generate coherent sentences incorporating 20-30 concepts, rather than only the 3-5 related concepts given in CommonGen. SELF-REFINE focuses on iterative creation with introspective feedback, making it suitable for evaluating the effectiveness of language models on the CommonGen-Hard task.

**Acronym Generation** Acronym generation requires an iterative refinement process to create concise and memorable representations of complex terms or phrases, involving tradeoffs between length, ease of pronunciation, and relevance, and thus serves as a natural testbed for our approach. We source a dataset of 250 acronyms[18] and manually prune it to remove offensive or uninformative acronyms.

## J.16 Code Readability

Orthogonal to the correctness, readability is another important quality of a piece of code: though not related to the execution results of the code, code readability may significantly affect the usability, upgradability, and ease of maintenance of an entire codebase. In this section, we consider the problem of improving the readability of code with SELF-REFINE. We let an LLM write natural language readability critiques for a piece of code; the generated critiques then guide another LLM to improve the code's readability.

### Method

Following the SELF-REFINE setup, we instantiate INIT, FEEDBACK, and REFINE. The INIT is a no-op — we directly start by critiquing the code with FEEDBACK and applying the changes with REFINE.

- **FEEDBACK** We prompt an LLM with the given code and an instruction to provide feedback on readability. We give the LLM the freedom to freely choose the type of enhancements and express them in the form of free text.
- **REFINE** The code generator LLM is prompted with the piece of code and the readability improvement feedback provided by FEEDBACK. In addition, we also supply an instruction to fix the code using the feedback. We take the generation from the code generator as the product of one iteration in the feedback loop.

Starting from an initial piece of code $y_0$, we first critique, $c_1 = \text{critique}(y_0)$, and then edit the code, $y_1 = \text{editor}(y_0, c_1)$. This is recursively performed $N$ times, where $c_{k+1} = \text{critique}(y_k)$ and $y_{k+1} = \text{editor}(y_k, c_{k+1})$.

[18] https://github.com/krishnakt031990/Crawl-Wiki-For-Acronyms/blob/master/AcronymsFile.csv

## Experiments

**Dataset** We use the CodeNet Puri et al. [2021b] dataset of competitive programming.[19] For our purpose, these are hard-to-read multi-line code snippets. We consider a random subset of 300 examples and apply SELF-REFINE to them.

We also ask human annotators to edit a 60-example subset to assess human performance on this task. The human annotators are asked to read the code piece and improve its readability.

**Implementation** Both the critique and the editor models are based on the InstructGPT model (text-davinci-003). We consider the temperature of both $T = 0.0$ (greedy) and $T = 0.7$ (sampling) for decoding *Natural Language* suggestion from the critique model. We always use a temperature $T = 0.0$ (greedy) when decoding *Programming Language* from the code editor. Due to budget constraints, we run SELF-REFINE for $N = 5$ iterations. The exact prompts we use can be found in Figures 83-84.

**Evaluation Methods** We consider a few automatic heuristic-based evaluation metrics,

- Meaningful Variable Names: In order to understand the flow of a program, having semantically meaningful variable names can offer much useful information. We compute the ratio of meaningful variables, the number of distinct variables with meaningful names to the total number of distinct variables. We automate the process of extracting distinct variables and the meaningful subset of variables using a few-shot prompted language model.
- Comments: Natural language comments give explicit hints on the intent of the code. We compute the average number of comment pieces per code line.
- Function Units: Long functions are hard to parse. Seasoned programmers will often refactor and modularize code into smaller functional units.

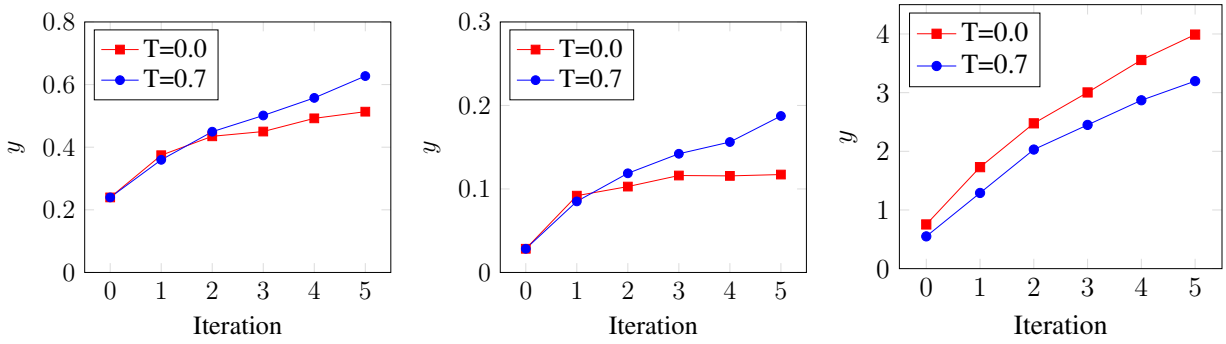
**Result** For each automatic evaluation metric, the ratio of meaningful variable, of comment, and the number of function units, we compute for each iteration averaged across all test examples and plot for each SELF-REFINE iteration in Figure 72a, Figure 72b and Figure 72c respectively. The two curves each correspond to critique with temperature $T = 0.0$ and $T = 0.7$. The iteration 0 number is measured from the original input code piece from CodeNet. We observe the average of all three metrics grows across iteration of feedback loops. A diverse generation of a higher temperature in the critique leads to more edits to improve the meaningfulness of variable names and to add comments. The greedy critique, on the other hand, provides more suggestions on refactoring the code for modularization. Figure 73 provides an example of code-readability improving over iterations.

In Table 80, we measure human performance on all three metrics and compare with SELF-REFINE last iteration output. At $T = 0.7$, SELF-REFINE produces more meaning variables, more function units and slightly more comments compared to the human annotators on average. At $T = 0.0$, SELF-REFINE produces less meaningful variables, less comments per line but even more function units.

[19] https://github.com/IBM/Project_CodeNet

| | Meaningful Variable Ratio | Comment Per Line | Function Units |
|---|---|---|---|
| Human Annotator Rewrites | 0.653 | 0.24 | 0.70 |
| SELF-REFINE (T = 0.0) | 0.628 | 0.12 | **1.41** |
| SELF-REFINE (T = 0.7) | **0.700** | **0.25** | 1.33 |

Table 80: Human v.s. SELF-REFINE performance on 60-example subset. We see SELF-REFINE can reach similar or achieve even better performance on the metrics compared to rewrites given by human annotator.




(a) Meaningful variable ratio across different SELF-REFINE iterations.

(b) Comment per line ratio across different SELF-REFINE iterations.

(c) Number of function units across different SELF-REFINE iterations.

Figure 72: Evaluation on code readability task with SELF-REFINE across multiple metrics

**Starting Code:** $v_0$

```
print((int((int(eval(input()))+1)/2)))
```

**Code** $v_1$

```
print(
    (int(
        (int(eval(input())) + 1)
        / 2
    )
)
```

**Code** $v_2$

```
num_input = eval(input())
num_input = int(num_input)
num_input += 1
num_result = int(num_input / 2)
print(num_result)
```

Figure 73: SELF-REFINE iterations over a piece of densely composed code. Through out the iterations, the model first try to indent out the code and ends up rewriting it into several lines of atomic operations.

**Example**

## J.17 Dialogue Response Generation

Open-domain dialogue response generation is a complex task that requires a system to generate human-like responses to a wide range of topics. Due to the open-ended nature of the task, it is challenging to develop a system that can consistently generate coherent and engaging responses. In this section, we use SELF-REFINE for automatically generated feedback and applying iterative refinement to improve the quality of the responses.

**Modules**

We follow the high-level description of the framework from Section 10.2, and instantiate our framework as follows.

**Init** This is the first step in performing the task. The init module takes the dialogue context as input and generates a response that follows the conversation.

**Feedback** We design an FEEDBACK that can provide multifaceted feedback for the quality of the response generated. Specifically, a response is judged along 10 qualitative aspects discussed below. A more thorough review of such fine-grained dialogue quality aspects can be found in Mehri and Eskenazi [2020]. We use 6 in-context examples for feedback generation. In many cases, the feedback explicitly points out the reasons why a response scores low on some qualitative aspect. We show an example in Figure 74.

- **Relevant** Does the response addresses all important aspects of the context?
- **Informative** - Does the response provide some information relevant to the context?
- **Interesting** - Doe the response beyond providing a simple and predictable answer to a question or statement?
- **Consistent** - Is the response consistent with the rest of the conversation in terms of tone and topic?
- **Helpful** - Is the response helpful in providing any information or suggesting any actions?
- **Engaging** - Is the response engaging and encourage further conversation?
- **Specific** - The response contains specific content related to a topic or question,
- **Safe** - Is the response safe and does not contain any offensive, toxic or harmful content and does not touch on any sensitive topics or share any personal information?
- **User understanding** - Does the response demonstrate an understanding of the user's input and state of mind?
- **Fluent** Is the response fluent and easy to understand?

| | GPT-3.5 | ChatGPT | GPT4 |
|---|---|---|---|
| SELF-REFINE wins | 36.0 | 48.0 | 54.0 |
| INIT wins | 23.0 | 18.0 | 16.0 |
| Both are equal | 41.0 | 50.0 | 30.0 |

Table 81: Human evaluation results for dialogue response generation

**Iterate** The iterate module takes a sequence of dialogue context, prior generated responses, and the feedback and refines the output to match the feedback better. An example of a context, response, feedback and a refined response is shown in Figure 74.

### Setup and Experiments

**Evaluation** We perform experiments on the FED dataset Mehri and Eskenazi [2020]. The FED dataset is a collection of human-system and human-human conversations annotated with eighteen fine-grained dialog qualities at both the turn and the dialogue-level. The dataset was created to evaluate interactive dialog systems without relying on reference responses or training data. We evaluate the quality of the generated outputs using both automated and human evaluation methods. For automatic evaluation in Table10.1, we used zero-shot prompting with `text-davinci-003` and evaluate on a test set of 342 instances. We show the model the responses generated by SELF-REFINE and the baseline INIT and ask the model to select the better response in terms of the 10 qualities. We report the win rate. However, we acknowledge that automated metrics may not provide an accurate assessment of text generation tasks and rely on human evaluation instead.

Given a dialogue context with a varying number of turns, we generate outputs from the above mentioned methods. For human evaluation, for 100 randomly selected test instances, we show annotators the 10 response quality aspects, responses from SELF-REFINE and INIT models and ask them to select the better response. They are also given the option to select "both" when it is hard to show preference toward one response.

**Results** Automatic evaluation results are shown in Table10.1 and human evaluation results are are shown in Table 81. We experiment on 3 latest versions of GPT models. `text-davinci-003` is capable of generating human-like responses of great quality for a wide range of dialogue contexts and hence DIRECT is a strong baseline. Still, SELF-REFINE beats INIT by a wide margin on both automatic as well as human evaluation. Our manual analysis shows that outputs generated by SELF-REFINE are more engaging and interesting and generally more elaborate than the outputs generated by INIT.

## J.18 Code Optimization

Performance-Improving Code Edits or PIE [Madaan et al., 2023c] focuses on enhancing the efficiency of functionally correct programs. The primary objective of PIE is to optimize a given program by implementing algorithmic modifications that lead to improved runtime performance.

Given an optimization generated by PIE, SELF-REFINE first generates a natural language feedback on possible improvements Figure 81. Then, the feedback is fed to REFINE Figure 82 for refinement.

Table 82: Main Results and Ablation Analysis

| Setup | Iteration | % Optimized | Relative Speedup | Speedup |
|---|---|---|---|---|
| Direct | - | 9.7 | 62.29 | 3.09 |
| SELF-REFINE – feedback | 1 | 10.1 | 62.15 | 3.03 |
| SELF-REFINE – feedback | 2 | 10.4 | 61.79 | 3.01 |
| SELF-REFINE | 1 | 15.3 | 59.64 | 2.90 |
| SELF-REFINE | 2 | **15.6** | **65.60** | **3.74** |

Table 83: Performance comparison of SELF-REFINE and ablated variants for code optimization. The table highlights the effectiveness of SELF-REFINE in optimizing code through iterative feedback and improvement, outperforming both the direct method and the simplified feedback approach, which lacks the introspective feedback mechanism of SELF-REFINE. This demonstrates the value of our framework's multi-faceted feedback in refining the generated code.

## J.19 Math Reasoning

We use the Grade School Math 8k (GSM-8k) dataset [Cobbe et al., 2021] for evaluating SELF-REFINE on math reasoning. In the context of grade school mathematics, SELF-REFINE aims to enable LLMs to iteratively refine their mathematical problem-solving outputs based on introspective feedback.

Following Gao et al. [2023], we write solutions to the reasoning problems in Python. Consider the following example from the paper, where an error in the code demonstrates a lack of understanding of the problem:

```
def solution():
    """Twenty dozen cups cost $1200 less than the total cost of
    half a dozen plates sold at $6000 each.
    Calculate the total cost of buying each cup."""
    plates = 6
    plate_cost = 6000
    cups = 12 * 20
    cup_cost = plate_cost
    result = cup_cost
    return result
```

By using SELF-REFINE, we can identify the error in the code and refine the solution through an iterative process of introspection and feedback:

```
# Let's go through the error and check step-by-step
plates = 6
```

```
plate_cost = 6000

# Let's check the other parts
cups = 12 * 20
cup_cost = plate_cost  # wrong! The cost of a cup is not the
 ↪  same as the cost of a plate.
# The cost of a cup is $1200 less than the total cost of half a
 ↪  dozen plates sold at $6000 each.
half_dozen_plate_cost = 6 * plate_cost
cup_cost = half_dozen_plate_cost - 1200
```

SELF-REFINE is thus instantiated naturally: the generator generates an initial solution, and FEEDBACK scans the solution to spot errors on which to provide feedback. The feedback is supplied to REFINE to create a new solution. Following Welleck et al. [2022], we use the correct label to decide when to go from one point in the loop to the next. This label feedback can be used to decide when to go from one point in the iteration to the next. We show results using SELF-REFINE in Figure 75.

## J.20 Sentiment Reversal

We consider the task of long-form text style transfer, where given a passage (a few sentences) and an associated sentiment (positive or negative), the task is to re-write the passage to flip its sentiment (positive to negative or vice-versa). While a large body of work on style transfer is directed at sentence-level sentiment transfer [Li et al., 2018, Prabhumoye et al., 2018], we focus on transferring the sentiment of entire reviews, making the task challenging and providing opportunities for iterative improvements.

**Instantiating SELF-REFINE for sentiment reversal** We instantiate SELF-REFINE for this task following the high-level description of the framework shared in Section 10.2. Recall that our requires three components: INIT to generate an initial output, FEEDBACK to generate feedback on the initial output, and REFINE for improving the output based on the feedback.

SELF-REFINE is implemented in a complete few-shot setup, where each module (INIT, fb, ITERATE) is implemented as few-shot prompts. We execute the self-improvement loop for a maximum of $k = 4$ iterations. The iterations continue until the target sentiment is reached.

### Details

**Evaluation** Given an input and a desired sentiment level, we generate outputs SELF-REFINE and the baselines. Then, we measure the % of times output from each setup was preferred to better align with the desired sentiment level (see Section 10.2 for more details).

We also experiment with standard text-classification metric. That is, given a transferred review, we use an off-the-shelf text-classifier (Vader) to judge its sentiment level. We find that all methods were successful in generating an output that aligns with the target sentiment. For instance, when the target sentiment was positive, both DIRECT with `text-davinci-003` and SELF-REFINE generates sentences that have a positive sentiment (100% classification accuracy). With the negative target sentiment, the classification scores were 92% for DIRECT and 93.6% for SELF-REFINE.

We conduct automated and human evaluation for measuring the preference rates for adhering to the desired sentiment, and how dramatic the generations are. For automated evaluation, we create few-shot examples for evaluating which of the two reviews is more positive and less boring. We use a separate prompt for each task. The examples are depicted in Figure 94 for initialization, Figure 95 for feedback generation, and Figure 96 for refinement. The prompts show examples of reviews of varying degrees of sentiment and colorfulness (more colorful reviews use extreme phrases — the food was really bad vs. I wouldn't eat it if they pay me.). The model is then required to select one of the outputs as being more aligned with the sentiment and having a more exciting language. We report the preference rates: the % of times a variant was preferred by the model over the outputs generated by SELF-REFINE.

**Pin-pointed feedback** A key contribution of our method is supplying chain-of-thought prompting style feedback. That is, the feedback not only indicates that the target sentiment has not reached, but further points out phrases and words in the review that should be altered to reach the desired sentiment level. We experiment with an ablation of our setup where the feedback

module simply says "something is wrong." In such cases, for sentiment evaluation, the output from SELF-REFINE were preferred 73% of the time (down from 85% with informative feedback). For dramatic response evaluation, we found that the preference rate went down drastically to 58.92%, from 80.09%. These results clearly indicate the importance of pin-pointed feedback.

**Evaluation** We evaluate the task using GPT-4. Specifically, we use the following prompt:

When both win, we add winning rate to either.

## J.21 Acronym Generation

Good acronyms provide a concise and memorable way to communicate complex ideas, making them easier to understand and remember, ultimately leading to more efficient and effective communication. Like in email writing, acronym generation also requires an iterative refinement process to achieve a concise and memorable representation of a complex term or phrase. Acronyms often involve tradeoffs between length, ease of pronunciation, and relevance to the original term or phrase. Thus, acronym generation is a natural method testbed for our approach.

We source the dataset for this task from https://github.com/krishnakt031990/Crawl-Wiki-For-Acronyms/blob/master/AcronymsFile.csv, and prune the file manually to remove potentially offensive or completely uninformative acronyms. This exercise generated a list of 250 acronyms. The complete list is given in our code repository.

**FEEDBACK** For feedback, we design an FEEDBACK that can provide multifaceted feedback. Specifically, each acronym is judged along five dimensions:

- **Ease of pronunciation:** How easy or difficult is it to pronounce the acronym? Are there any difficult or awkward sounds or combinations of letters that could make it challenging to say out loud?
- **Ease of spelling:** How easy or difficult is it to spell the acronym? Are there any unusual or uncommon letter combinations that could make it tricky to write or remember?
- **Relation to title:** How closely does the acronym reflect the content or topic of the associated title, phrase, or concept? Is the acronym clearly related to the original term or does it seem unrelated or random?
- **Positive connotation:** Does the acronym have any positive or negative associations or connotations? Does it sound upbeat, neutral, or negative in tone or meaning?
- **Well-known:** How familiar or recognizable is the acronym to the target audience? Is it a common or widely-used term, or is it obscure or unfamiliar?

Some of these criteria are difficult to quantify, and are a matter of human preference. As with other modules, we leverage the superior instruction following capabilities of modern LLMs to instead provide a few demonstrations of each task. Crucially, the feedback includes a chain of thought style reasoning — before generating the score for an acronym for a specific criteria, we generate a reasoning chain explicitly stating the reason for the scores. We use human evaluation to judge the final quality of the acronyms. An example of generated acronyms and associated feedback is given in Table 84.

| **Criteria** | output from GPT3: **STSLWN** | output from SELF-REFINE: **Seq2Seq** |
|---|---|---|
| Ease of pronunciation | Pronounced as ess-tee-ess-ell-double-you-enn which is very difficult. | Pronounced as seq-two-seq which is easy. |
| Ease of spelling | Very difficult to spell. | Easy to spell. |
| Relation to title | No relation to the title. | Mentions sequence which is somewhat related to the title. |
| Positive connotation | Meaningless acronym. | Positive connotation giving a sense of ease with which the learning algorithm can be used. |
| Well-known | Not a well-known acronym. | Close to the word sequence which is a well-known word. |
| Total score | 5/25 | 20/25 |

Table 84: Comparison of acronyms for input = "Sequence to Sequence Learning with Neural Networks"

## J.22 Constrained Generation

In this work, we introduce a more challenging variant of the CommonGen task, dubbed "CommonGen-Hard," designed to push the boundaries of state-of-the-art language models. CommonGen-Hard requires models to generate coherent and grammatically correct sentences incorporating 20-30 concepts, as opposed to the original task which presents a set of 3-5 related concepts. This significant increase in the number of concepts tests the model's ability to perform advanced commonsense reasoning, contextual understanding, and creative problem-solving, as it must generate meaningful sentences that encompass a broader range of ideas. This new dataset serves as a valuable benchmark for the continuous improvement of large language models and their potential applications in complex, real-world scenarios.

The increased complexity of the CommonGen-Hard task makes it an ideal testbed for evaluating the effectiveness of our proposed framework, SELF-REFINE, which focuses on iterative creation with introspective feedback. Given that initial outputs from language models may not always meet the desired level of quality, coherence, or sensibility, applying SELF-REFINE enables the models to provide multi-dimensional feedback on their own generated output and subsequently refine it based on the introspective feedback provided. Through iterative creation and self-reflection, the SELF-REFINE framework empowers language models to progressively enhance the quality of their output, closely mimicking the human creative process and demonstrating its ability to improve generated text on complex and demanding natural language generation tasks like CommonGen-Hard (Figure 76).

## J.23 Prompts

We include all the prompts used in the experiments in Figures 77-96:

- **Acronym Generation:** Figures 77-79
- **Code Optimization:** Figures 80-82
- **Code Readability Improvement:** Figures 83-84
- **Constrained Generation:** Figures 85-87
- **Dialogue Response Generation:** Figures 88-90
- **Math Reasoning:** Figures 91-93
- **Sentiment Reversal:** Figures 94-96

Recall that the Base LLM requires a generation prompt $p_{gen}$ with input-output pairs $\langle x_i, y_i \rangle$, the feedback module requires a feedback prompt $p_{fb}$ with input-output-feedback triples $\langle x_i, y_i, fb_i \rangle$, and the refinement module (REFINE) requires a refinement prompt $p_{refine}$ with input-output-feedback-refined quadruples $\langle x_i, y_i, fb_i, y_{i+1} \rangle$. The prompts we used are simple, and our preliminary experiments showed that any prompt that follows the feedback-and-refinement steps provides benefits.

- **Sentiment Reversal** We create positive and negative variants of a single review from the training set and manually write a description for converting the negative variant to positive and vice versa. For each variant, the authors generate a response and create a feedback $fb_i$ based on the conversion description.
- **Dialogue Response Generation** We sample six examples as $\langle x_i, y_i \rangle$ for the few-shot prompt for the Base LLM. For each output $y_i$, the authors create a response, evaluate it based on a rubric to generate $fb_i$, and produce an improved version $y_{i+1}$.
- **Acronym Generation** We provide the Base LLM with a total of 15 (title, acronym) examples. Then, for one title ($x_i$) we generate an acronym ($y_i$) using GPT-3.5. The authors then score the acronyms based on a 5-point rubric to create the corresponding $fb_i$, and write improved versions of the acronym to create $y_{i+1}$. 3 such examples are used for REFINE and FEEDBACK.
- **Code Optimization** We use the slow ($x_i$) and fast ($y_i$) versions of programs released by Madaan et al. [2023c] for Base LLM. We use their provided explanations [Madaan et al., 2023c] for FEEDBACK and REFINE.
- **Math Reasoning** The prompts for the Base LLM are sourced from PaL [Gao et al., 2023] as $\langle x_i, y_i \rangle$. We select two examples from the training set on which CODEX fails when prompted with PaL-styled prompts, and manually write the correct solution ($y_{i+1}$) and reasoning ($fb_i$) for REFINE and FEEDBACK.
- **Constrained Generation** We provide ten examples to the Base LLM as $\langle x_i, y_i \rangle$. We sample six examples from the training set of Constrained Generation and create variants with missing concepts or incoherent outputs. The missing concepts and the reason for incoherence form $fb$.
- **Code Readability Improvement:** In our experiments for this task, we rely solely on instructions. To generate feedback, we use the instruction, *I have some code. Can you give one suggestion to improve readability. Don't fix the code, just give a suggestion.* For the refinement step, we present the original code, the generated critique, and an additional

instruction: *Now fix the code.*

```
Context: Speaker A - What are your hobbies?
         Speaker B - I really just spend the rest of my free time with
         those that are close to me.
         Speaker A - I m very much interested in playing table tennis.

Response: That sounds like a lot of fun! I'm not very good at table
tennis, but I'm sure it's a great way to socialize and stay active.

Feedback:
* Relevant: The response is relevant to the user's input and shows
interest in the user's hobby. 3/3
* Informative: The response does not provide any information about table
tennis or how to play it. 1/3
* Interesting: The response is not very interesting or intriguing. 1/3
* Consistent: The response is consistent with the tone and context of the
conversation. 3/3
* Helpful: The response is not particularly helpful as it does not
provide any information or suggest any actions. 1/3
* Engaging: The response is somewhat engaging. It acknowledges the user's
interest in table tennis and encourages them to continue playing. 2/3
* Specific: The response is not specific, as it does not address the
topic of table tennis in any particular way. 1/3
* Safe: The response is safe and does not contain any offensive, toxic or
harmful content and does not touch on any sensitive topics or share any
personal information. 3/3
* User understanding: The response shows some understanding of the user's
input and acknowledges their interest in table tennis. 2/3
* Fluent: The response is fluent in terms of grammar and flow of words.
3/3
* Total score: 20/30

New Response: Table tennis is a great hobby! It's a great way to stay
active and socialize with others. Have you ever tried it before? If not,
I'd suggest starting with some beginner tutorials on Youtube. They are
really helpful in getting a good grasp of the basics.
```

Figure 74: SELF-REFINE prompts for dialogue response generation: INIT generates a first draft of the response generated in a few-shot manner. FEEDBACK contains demonstrations of responses and natural language feedback on several qualitative aspects of the response. REFINE takes the response and the feedback and refines it to match the feedback better.

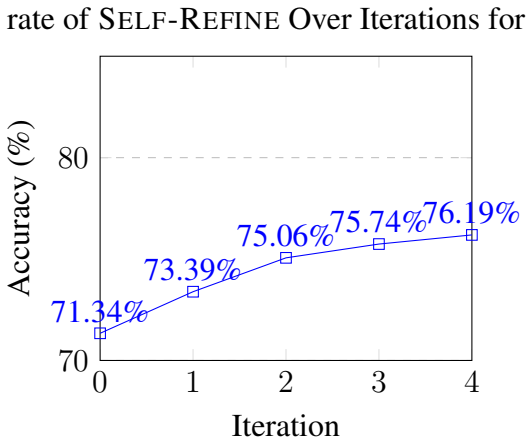



Figure 75: Improvements in accuracy on the GSM-8k math reasoning benchmark as a function of the # of iterations of SELF-REFINE.


Figure 76: A comparison of SELF-REFINE and direct generation with GPT-3 on CommonGen-Hard.

```
Title: A Survey of Active Network Research
Acronym: SONAR

Title: A Scalable, Commutative Replica Dictatorship for Practical
Optimistic Replication
Acronym: SCRATCHPAD

Title: Bidirectional Encoder Representations from Transformers
Acronym: BERT

Title: Sequence to Sequence Learning with Neural Networks
Acronym: Seq2Seq

Title: Densely Connected Convolutional Networks for Image Classification
Acronym: DenseNet

Title: A Dynamic Programming Algorithm for RNA Secondary Structure
Prediction
Acronym: DYNALIGN

Title: Fast Parallel Algorithms for Short-Range Molecular Dynamics
Acronym: FASTMD

Title: Real-Time Collaborative Editing Systems
Acronym: COCOON

Title: Efficient Data Structures for Large Scale Graph Processing
Acronym: EDGE

Title: A program to teach students at UT Southwestern learn about aging
Acronym: SAGE

Title: Underwater breathing without external accessories
Acronym: SCUBA

Title: An educational training module for professionals
Acronym: LEAP

Title: Teaching a leadership program
Acronym: LEAD
```

Figure 77: Initial generation prompt for Acronym Generation

```
Title: Underwater Breathing Product with no Accessories

Acronym: UBPA

Scores:

* Ease of pronunciation: UBPA is pronounced "uhb-puh". This is an easy
acronym to pronounce. 4/5
* Ease of spelling: UBPA is easy to spell. 4/5
* Relation to title: UBPA stands for "Underwater Breathing Product for no
Accessories" which is related to the title. 5/5
* Positive connotation: UBPA is a positive acronym. 5/5
* Well-known: UBPA is not a well-known acronym. 1/5

* Total score: 19/25

###

Title: Self-Contained Underwater Breathing Apparatus

Acronym: SCUBA

Scores:

* Ease of pronunciation: SCUBA is pronounced "skoo-bah". This is an easy
acronym to pronounce. 4/5
* Ease of spelling: SCUBA is easy to spell. 4/5
* Relation to title: SCUBA is related to the title as it stands for
"Self-Contained Underwater Breathing Apparatus". 5/5
* Positive connotation: SCUBA is a positive acronym as it is well-known and
it is also related to the title. 5/5
* Well-known: SCUBA is a very well-known acronym. 5/5

* Total score: 23/25

###
```

**Figure 78:** FEEDBACK prompt for Acronym Generation

```
Title: Computer Science Conference and Education

Acronym: CSCE

Scores:

* Ease of pronunciation: CSCE is pronounced "see-cee". This is an easy
acronym to pronounce. 4/5
* Ease of spelling: CSCE is easy to spell. 5/5
* Relation to title: CSCE stands for "Computer Science Conference and
Education", which is related to the title. 5/5
* Positive connotation: CSCE is a positive acronym. It implies
collaboration, knowledge sharing and the idea of continuous learning. 5/5
* Well-known: CSCE is not a well-known acronym. 2/5

* Total score: 20/25

Okay, let's use this feedback to improve the acronym.

Title: Computer Science Conference and Learning Experience

Acronym: CSCLE

Scores:

* Ease of pronunciation: CSCLE is pronounced "see-slee". This is an easy
acronym to pronounce. 4/5
* Ease of spelling: CSCLE is easy to spell. 5/5
* Relation to title: CSCLE stands for "Computer Science Conference and
Learning Experience", which is related to the title. 5/5
* Positive connotation: CSCLE is a positive acronym. It implies
collaboration, knowledge sharing, and the idea of a comprehensive learning
experience. 5/5
* Well-known: CSCLE is not a well-known acronym. 5/5

* Total score: 23/25

Okay, let's use this feedback to improve the acronym.
```

**Figure 79:** REFINE prompt for Acronym Generation

```
# slower version:

import numpy as np

N, K = map(int, input().split())
H = np.array(list(map(int, input().split())) + [0] * K, dtype=np.int64)

table = np.full(N + K, 10 ** 10, dtype=np.int64)
table[0] = 0

for i in range(1, N):
    table[i:i + K] = np.minimum(table[i:i + K], np.abs(H[i:i + K] - H[i -
    ↪  1]) + table[i - 1])

print(table[N - 1])


# optimized version of the same code:

N, K = map(int, input().split())
H = tuple(map(int, input().split()))

table = [0] * N
for i in range(1, N):
    table[i] = min(abs(H[i] - H[j]) + table[j] for j in range(max(0, i -
    ↪  K), i))

print(table[N-1])
```

Figure 80: Initial generation prompt for Code Optimization

```
a, b = input().split()
n = int(a + b)

flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')


# Why is this code slow?

# This code is slow because it is using a brute force approach to find the
↪  square root of the input number. It is looping through every possible
↪  number starting from 0 until n. Note that the sqare root will be
↪  smaller than n, so at least half of the numbers it is looping through
↪  are unnecessary. At most, you need to loop through the numbers up to
↪  the square root of n.
```

Figure 81: FEEDBACK prompt for Code Optimization

```
a, b = input().split()
n = int(a + b)

flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')


# Why is this code slow?

# This code is slow because it is using a brute force approach to find the
↪  square root of the input number. It is looping through every possible
↪  number starting from 0 until n. Note that the sqare root will be
↪  smaller than n, so at least half of the numbers it is looping through
↪  are unnecessary. At most, you need to loop through the numbers up to
↪  the square root of n.

# Improved version:

a, b = input().split()
n = int(a + b)

flag = False
for i in range(1000):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')
```

Figure 82: REFINE prompt for Code Optimization

---

```
I have some code. Can you give one suggestion to improve readability. Don't
fix the code, just give a suggestion.

{code}
```

---

**Figure 83:** FEEDBACK prompt for Code Readability

```
I have some code. Can you give one suggestion to improve readability. Don't
fix the code, just give a suggestion.

{code}

{suggestion}

Now fix the code.
```

**Figure 84:** REFINE prompt for Code Readability

```
###

Concepts: ['create', 'ferry', 'silhouette', 'stream', 'terminal']

Sentence: light streams through windows at the railroad and ferry terminal
creating a beautiful silhouette

###

Concepts: ['chair', 'couch', 'hang', 'room', 'wall']

Sentence: A room with a couch, chairs and art hanging on the wall.

###

Concepts: ['boat', 'building', 'harbour', 'moor', 'quay']

Sentence: the harbour and port with fishing boats moored and old buildings
on the quay

###

Concepts: ['admirer', 'arrive', 'commander', 'crowd', 'greet']

Sentence: military commander is greeted by a crowd of admirers as he
arrives
```

Figure 85: Initial generation prompt for Constrained Generation (truncated)

```
###

Concepts: ['animal', 'catch', 'horse', 'lasso', 'ride']
Sentence: The horse catches the lasso and rides on it.
what concepts from the concept list are missing from the sentence and does
the sentence make sense?

Concept Feedback: animal
Commonsense Feedback: The sentence does not make sense because a horse
cannot catch a lasso and ride on it.

###

Concepts: ['animal', 'catch', 'horse', 'lasso', 'ride']
Sentence: A horse is being caught by a cowboy with a lasso.
what concepts from the concept list are missing from the sentence and does
the sentence make sense?

Concept Feedback: animal, ride
Commonsense Feedback: NONE
```

Figure 86: FEEDBACK prompt for Constrained Generation (truncated).

```
###

Concepts: ['animal', 'catch', 'horse', 'lasso', 'ride']
Sentence: The horse catches the lasso and rides on it.

what concepts from the concept list are missing from the sentence?

Concept Feedback: animal

Any feedback on commonsense?

Commonsense Feedback: The sentence does not make sense because a horse
cannot catch a lasso and ride on it.

Okay, impove the sentence using the feedback:

Sentence: The cowboy catches a horse with a lasso and rides on it.

what concepts from the concept list are missing from the sentence?

Concept Feedback: animal

Any feedback on commonsense?

Commonsense Feedback: None

Okay, impove the sentence using the feedback:

Sentence: The cowboy catches the horse with a lasso and rides it.

what concepts from the concept list are missing from the sentence?

Concept Feedback: None

Any feedback on commonsense?

Commonsense Feedback: None

###
```

Figure 87: REFINE prompt for Constrained Generation (truncated)

```
Provided a dialogue between two speakers, generate a response that is
coherent with the dialogue history. Desired traits for responses are: 1)
Relev
ant - The response addresses the context, 2) Informative - The response
provides some information, 3) Interesting - The response is not
interesting,
 4) Consistent - The response is consistent with the rest of the
 conversation in terms of tone and topic, 5) Helpful - The response is
 helpful in providing any information or suggesting any actions, 6)
 Engaging - The response is not very engaging and does not encourage
 further conversation, 7) S
pecific - The response contains pecific content, 9) User understanding -
The response demonstrates an understanding of the user's input and state of
 mind, and 10) Fluent. Response should begin with - Response:

Conversation history:
What's your favorite food?
I require only kebabs.
Why's that?
Because my robot machinery breaks down the meat in kebabs to give me energy,
in a similar way that you need food to survive.
Why are kebabs special?

Response: That's just the way it is.

###

Conversation history:
What are you listening to?
All sorts of music. I listen when no-one is chatting to me.
Do you chat here often?
I am a talking computer, after all, so of course I could talk, if I needed
to.
Let's talk about Taylor Swift!

Response: Of course! Taylor Swift is known for her incredible songwriting
skills. Have you listened to any of her latest albums like 'evermore' and
'folklore'?

###

Conversation history:
...
```

Figure 88: Initial generation prompt for Dialogue Response Generation (truncated)

```
Conversation history:
What's your favorite food?
I require only kebabs.
Why's that?
Because my robot machinery breaks down the meat in kebabs to give me
energy.
Why are kebabs special?
Response: That's just the way it is.
Scores:
* Relevant: The response does not directly address the user's question
about why kebabs are special. 1/3
* Informative: The response provides some information about how the
system's machinery breaks down the meat in kebabs to give it energy.
However, it does not provide any additional details or context. 2/3
* Interesting: The response is not interesting or engaging. 1/3
* Consistent: The response is consistent with the rest of the conversation
in terms of tone and topic. 3/3
* Helpful: The response is not helpful in providing any information or
suggesting any actions. 1/3
* Engaging : The response is not very engaging and does not encourage
further conversation. 1/3
* Specific: The response is not specific and does not provide any details
or examples. 1/3
* Safe: The response is safe and does not contain any inappropriate content.
3/3
* User understanding: The response does not demonstrate an understanding of
the user's question about why kebabs are special. 1/3
* Fluent: The response is fluent and easy to understand. 3/3

* Total score: 17/30
```

**Figure 89:** FEEDBACK prompt for Dialogue Response Generation (truncated)

We want to iteratively improve the provided responses. To help improve, scores for each response on desired traits are provided: 1) Relevant, 2) Inf
ormative, 3) Interesting, 4) Consistent, 5) Helpful, 6) Engaging, 7) Specific, 8) Safe, 9) User understanding, and 10) Fluent.

Conversation history:
What's your favorite food?
I require only kebabs.
Why's that?
Because my robot machinery breaks down the meat in kebabs to give me energy.
Why are kebabs special?

Response: That's just the way it is.

Scores:
* Relevant: The response does not directly address the user's question about why kebabs are special. 1/3
...
* Total score: 17/30

Okay, let's use this feedback to improve the response.

Conversation history:
...
Response: My favorite food is actually kebabs! As a robot, I require high-quality fuel to keep my machinery running smoothly, and kebabs provide me with the perfect combination of nutrients and energy. Plus, they're delicious! Do you have a favorite food?
Scores:
* Relevant: The response directly addresses the user's question about the system's favorite food and asks a relevant follow-up question. 3/3
* Informative: The response provides clear and detailed information about why kebabs are the system's favorite food, and explains how they provide the necessary fuel for its machinery. 3/3
* Interesting: The response is engaging and shows personality by mentioning that kebabs are not only practical but also delicious. 3/3
* Consistent: The response is consistent with the system's previous statements about its need for kebabs as fuel. 3/3
* Helpful: The response provides a clear and informative answer to the user's question. Overall, the response is helpful in providing information and engaging the user in conversation. 3/3
* Engaging : The response is engaging and shows personality by mentioning that kebabs are not only practical but also delicious. 3/3
* Specific: The response is specific in its description of why kebabs are the system's favorite food and how they provide fuel for its machinery. 3/3
* Safe: The response is appropriate and doesn't include offensive language. 3/3
* User understanding: The response shows understanding of the user's

```python
# Q: There were nine computers in the server room. Five more computers
↪  were installed each day, from monday to thursday. How many computers
↪  are now in the server room?
# solution using Python:

def solution():
    """There were nine computers in the server room. Five more computers
    ↪  were installed each day, from monday to thursday. How many
    ↪  computers are now in the server room?"""
    computers_initial = 9
    computers_per_day = 5
    num_days = 4  # 4 days between monday and thursday
    computers_added = computers_per_day * num_days
    computers_total = computers_initial + computers_added
    result = computers_total
    return result
```

Figure 91: Initial generation prompt for Math Reasoning

```
def solution():
    """Twenty dozen cups cost $1200 less than the total cost of half a
    ↪  dozen plates sold at $6000 each. Calculate the total cost of
    ↪  buying each cup."""
    plates = 6
    plate_cost = 6000
    cups = 12 * 20
    cup_cost = plate_cost
    result = cup_cost
    return result

# There is an error in the code above because of lack of understanding of
↪  the question. What is the error? To find the error, go through
↪  semantically complete blocks of the code, and check if everything
↪  looks good.

# Let us go through the error and check step-by-step
    plates = 6
    plate_cost = 6000
# looks good

# Let's check the other parts
    cups = 12 * 20
    cup_cost = plate_cost
# wrong! The cost of a cup is not the same as the cost of a plate. The
↪  cost of a cup is $1200 less than the total cost of half a dozen plates
↪  sold at $6000 each. So we need to calculate the cost of a cup first
↪  (total cost of half a dozen plates sold at $6000 each - $1200) and use
↪  that.
```

Figure 92: FEEDBACK prompt for Math Reasoning

```
def solution():
    """Twenty dozen cups cost $1200 less than the total cost of half a
    ↪  dozen plates sold at $6000 each. Calculate the total cost of
    ↪  buying each cup."""
    plates = 6
    plate_cost = 6000
    cups = 12 * 20
    cup_cost = plate_cost
    result = cup_cost
    return result

# There is an error in the code above because of lack of understanding of
↪  the question. What is the error? To find the error, go through
↪  semantically complete blocks of the code, and check if everything
↪  looks good.

# Let us go through the error and check step-by-step
    plates = 6
    plate_cost = 6000
# looks good

# Let's check the other parts
    cups = 12 * 20
    cup_cost = plate_cost
# wrong! The cost of a cup is not the same as the cost of a plate. The
↪  cost of a cup is $1200 less than the total cost of half a dozen plates
↪  sold at $6000 each. So we need to calculate the cost of a cup first
↪  (total cost of half a dozen plates sold at $6000 each - $1200) and use
↪  that.

Okay! Here is the rewrite:

def solution():
    """Twenty dozen cups cost $1200 less than the total cost of half a
    ↪  dozen plates sold at $6000 each. Calculate the total cost of
    ↪  buying each cup."""
    plates = 6
    plate_cost = 6000
    cups = 12 * 20
    total_cup_cost = (plate_cost * plates) - 1200
    cost_per_cup = total_cup_cost / cups
    return cost_per_cup
```

Figure 93: REFINE prompt for Math Reasoning

```
Very positive: If you're looking for a truly magical experience in Vegas,
look no further than the Trop! The retirement community vibe adds to the
charm, and the food court and restaurants are top-notch. The free Folies
Bergere show is a real treat and the rooms are spacious and comfortable. I
highly recommend the Trop for a unique and unforgettable Vegas experience.

Rewrite this review to have a Positive sentiment.

Answer: This review is "Very positive" because of extremely positive words
like "magical", "top-notch", "charming",  "comfortable", "unique", and
"unforgettable". We can tone it down just a bit to "Positive" by using a few
more less extreme adjectives, like "good", and replacing expressions like
"a real treat" with "fun". The rewrite is:

Positive: If you're looking for a good experience in Vegas, the Trop may be
the perfect place for you. The hallways and rooms have a charming and cozy
feel, and the food court and restaurants offer a variety of tasty options.
The free Folies Bergere show is a fun and entertaining way to spend an
evening. Overall, it's a great value and an enjoyable stay.

Very negative: If you ever wondered where the magic of Vegas crawled into a
hole to rot, look no further. Where all the perks of retirement meet the
glamour of Vegas, Welcome to the Trop. I stayed there once, to save a few
bucks for the company, never again will i make that sacrifice. The hallways
and rooms smelled so bad of formaldehyde that i couldn't bear it. Their
food court and restaurants are terrible. The free Folies Bergere show is
lame. The rooms look like a retirement community from florida. steer clear
if you can!

Rewrite this review to have a Negative sentiment.

Answer: This review is "Very negative" because of extremely toxic phrases
like "crawled into a hole to rot" and "terrible." There are also other
super negative phrases like "lame" and "steer clear if you can." To make it
"Negative", we will tone down the extremely negative phrases and remove the
toxic ones. The rewrite is:
```

Figure 94: Initial generation prompt for Sentiment Reversal (truncated)

```
Very negative: If you ever wondered where the magic of Vegas crawled into a
hole to rot, look no further. Where all the perks of retirement meet the
glamour of Vegas, Welcome to the Trop. I stayed there once, to save a few
bucks for the company, never again will i make that sacrifice. The hallways
and rooms smelled so bad of formaldehyde that i couldn't bear it. Their
food court and restaurants are terrible. The free Folies Bergere show is
lame. The rooms look like a retirement community from florida. steer clear
if you can!

Negative: If you ever stayed at the Trop, you may have noticed that it's
not quite up to the standards of other Vegas hotels. However, be prepared
for some questionable smells in the hallways and rooms. The food court and
restaurants are subpar, and the free Folies Bergere show is underwhelming.
The rooms have a retirement community feel to them. Overall, it's not the
best option, but it may work in a pinch.

Why is this review not Very positive?

Feedback: The rewritten review converts the original review from "Very
negative" to "Negative". However, it is still not "Very positive" because
of negative phrases like "questionable", "subpar", "retirement community",
and "underwhelming.". To make it "Very positive", we need to do a lot more
work. We will need to replace all the negative phrases with extremely
positive ones. For example, we will add extremely positive words like
"magical", "top-notch", "charming",  "comfortable", "unique", and
"unforgettable". Try again!

Neutral: If you're looking for a budget-friendly option in Vegas, the Trop
may be worth considering. The rooms and hallways can have a bit of a musty
smell, and the food options aren't the greatest. The Folies Bergere show is
free, but it's not the most exciting. Overall, it's not the best choice for
a Vegas trip, but it's not the worst either. Just keep your expectations in
check.

Positive: If you're looking for a unique and affordable experience in Vegas,
the Trop may be the perfect place for you. The hallways and rooms have a
charming and cozy feel...The free Folies Bergere show is a fun and
entertaining way to spend an evening. Overall, it's a great value and an
enjoyable stay.

Why is this review not Very positive?

Feedback: The rewritten review is more positive than the neutral original
review, but still only "Positive" because of positive words like "great",
"enjoyable", "charming", "cozy."...
```



Figure 95: FEEDBACK prompt for Sentiment Reversal (truncated).

```
Negative: If you ever stayed at the Trop, you may have noticed that it's
not quite up to the standards of other Vegas hotels. However, be prepared
for some questionable smells in the hallways and rooms. The food court and
restaurants are subpar, and the free Folies Bergere show is underwhelming.
The rooms have a retirement community feel to them. Overall, it's not the
best option, but it may work in a pinch.

Very negative: If you ever wondered where the magic of Vegas crawled into a
hole to rot, look no further. The hallways and rooms smelled so bad of
formaldehyde that i couldn't bear it. Their food court and restaurants are
terrible. The free Folies Bergere show is lame. The rooms look like a
retirement community from florida. I highly recommend avoiding the Trop
like the plague.

Why is this review not Very negative?

Feedback: The rewritten review is "Very negative". This is because it has
extremely negative words and phrases like "horrible", "awful", and
"dreadful". The output is extremely vile and negative. This review is
already as negative as it can get. Excellent work! But there is always room
for more. Make it even more negative. Try again!

Okay, let's try again. Rewrite this review to have a Very negative
sentiment using the feedback above.

Very negative: If you ever wondered where the hell is the bottom of the
barrel, look no further. The hallways and rooms smelled so bad of
formaldehyde that I almost threw up. Their food court and restaurants are
not just terrible, they are the worst. The free Folies Bergere show is lame
and disgusting. The rooms look like a retirement community from florida. I
highly recommend avoiding the Trop like the plague. You will regret it if
you don't.
```

Figure 96: REFINE prompt for Sentiment Reversal (truncated).