Thesis Proposal

Enhancing Language Models with Structured Reasoning

Aman Madaan

April 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 out-of-the-box solutions typically involve leveraging the Seq2Seq paradigm, where text-based input and output are the norm. While this approach provides a convenient foundation for many tasks, practical deployments demand solutions capable of addressing the shortcomings such as brittleness when handling complex problems, the absence of feedback mechanisms, and an inherent black-box nature hindering model interpretability.

This thesis proposes to address these limitations and enhance contemporary language models by integrating structured elements into their design and operation. Structure, in this context, is defined as the organization and representation of data in a systematic, hierarchical, or relational way, coupled with incorporating structural elements or 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 to represent the input. 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. Including structure 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.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

The widespread availability of user-friendly libraries for text-generation and reasoning has revolutionized the adaptation of numerous tasks within both the seq2seq framework and next-token prediction approaches [Radford, 2018, Raffel et al., 2020a, Yangfeng Ji and Celikyilmaz, 2020]. This extends beyond tasks innately suited to these paradigms, such as dialogue generation and summarization [Zhang et al., 2020b, Gehrmann et al., 2021], 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., 2021a, 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., 2017a, Wolf et al., 2019] can sometimes lead to overlooking the inherent trade-offs and limitations associated with using such out-of-the-box solutions. With a simple TSV file containing two columns for inputs and outputs, data can be seamlessly converted into strings, facilitating rapid implementation and experimentation. However, this powerful and straightforward setup comes with certain trade-offs.

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 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

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

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., 2016], 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.

1.1.2 Infusing Structure: our contribution

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 concentrate on structure as it pertains to the organization of data utilized for training [Bengio et al., 2013, Schmidhuber, 2015], as well as leveraging structure during the inference process to enhance both training and inference outcomes [Vaswani et al., 2017, Devlin et al., 2019, Lake et al., 2017].

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

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., 2014, 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.

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.

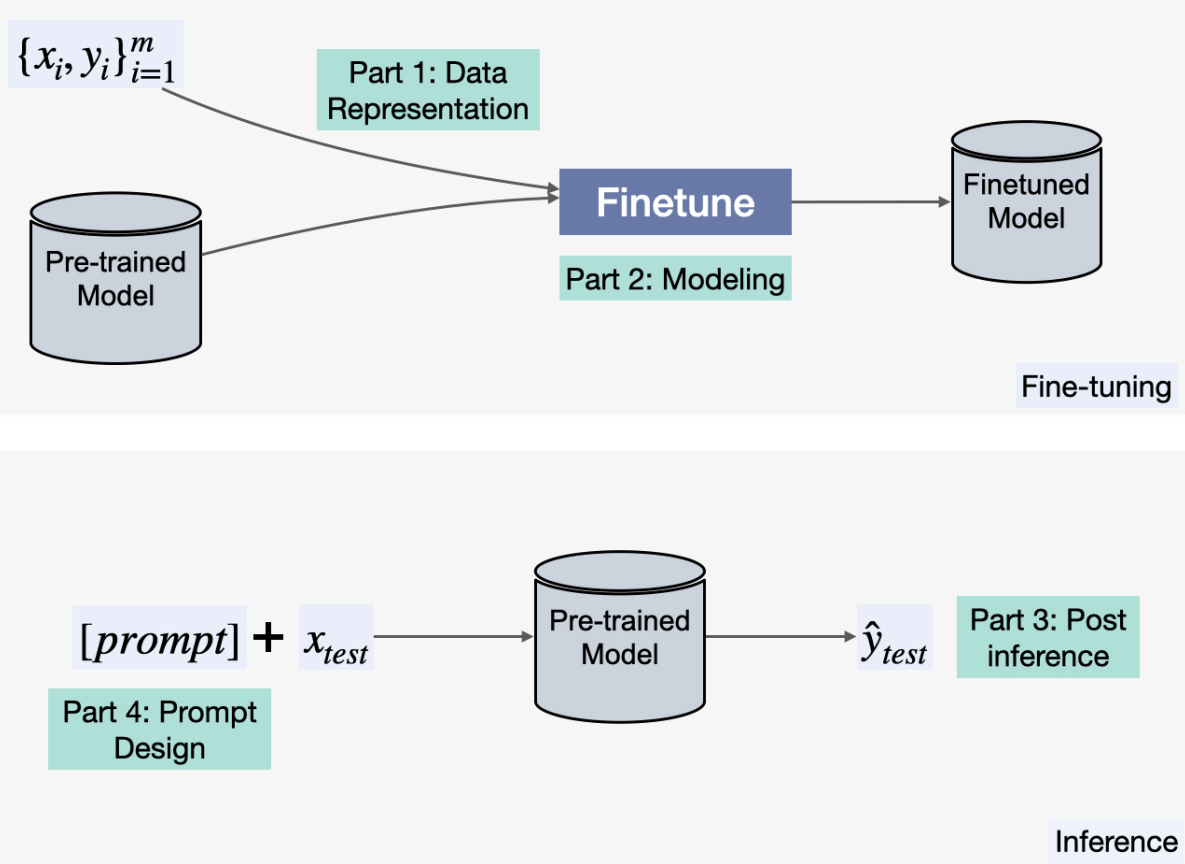
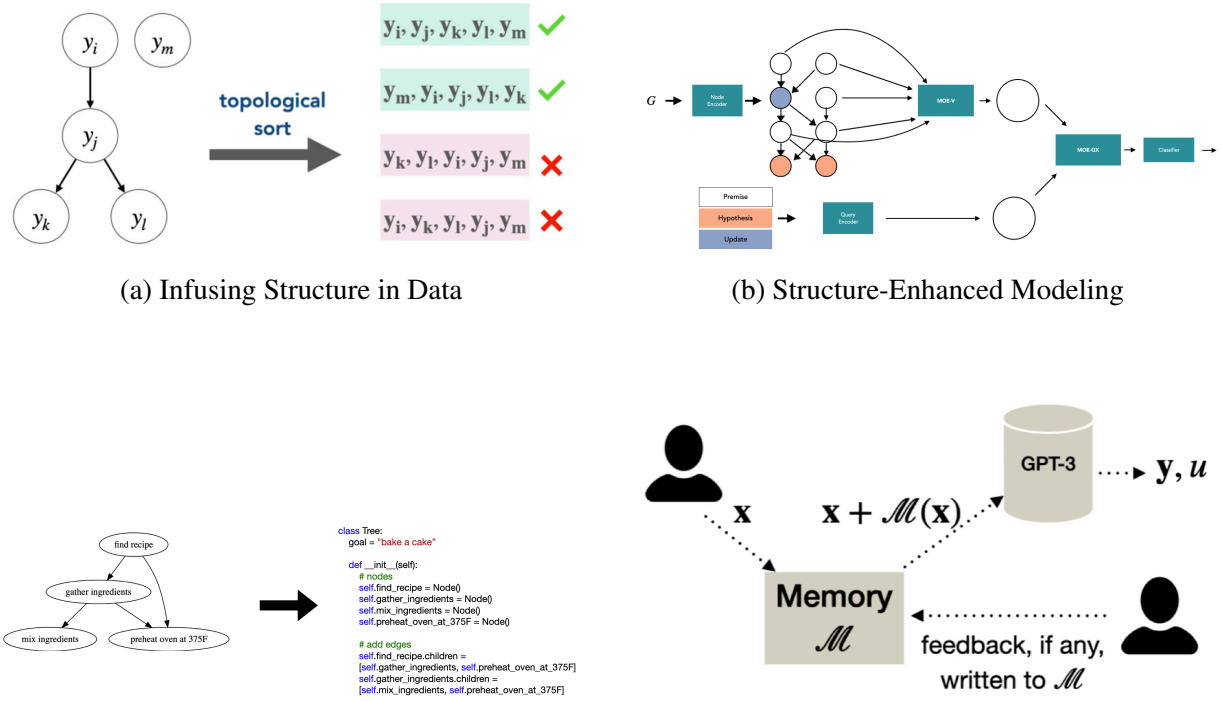


Figure 1.1: Overview of this thesis proposal

- **Part II: Structure-Assisted Modeling** delves into structure-enhanced generation and reasoning.
  - Chapter 4 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 5 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.
- **Part III: Leveraging Structure during Inference** explores approaches in graph generation, structured commonsense reasoning, and program-aided language models.
  - Chapter 6 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.
- **Part IV: Post-Inference Enhancements for LLMs** examines two chapters focused on enhancing large language models (LLMs) through user interactions and iterative refinement.



(a) Infusing Structure in Data

(b) Structure-Enhanced Modeling

(c) Exploiting Structure during Inference

(d) Post-Inference LLM Enhancements

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.

- Chapter 7 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.

Part	Work	Status
Part I	Event-level temporal graph generation	**NAACL 2021**
	Conditional set generation (SETAUG)	**EMNLP 2022**
	Performance-Improving Code Edits	In Progress
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	In Progress

Table 1.1: Thesis Status

1.3 Proposed Timeline

The proposed timeline to complete the thesis is as follows:

- June 2023 - Thesis proposal
- May 2023 - August 2023 Summer internship.
- September 2023 - December 2023 Finish proposed work.
- Jan 2024 - March 2024: Job search.
- April 2024 - Thesis Defense

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 **??**
2. Conditional Set Generation using SEQ2SEQ models.

Chapter 2

Neural Language Modeling for Contextualized Temporal Graph Generation

2.1 Introduction

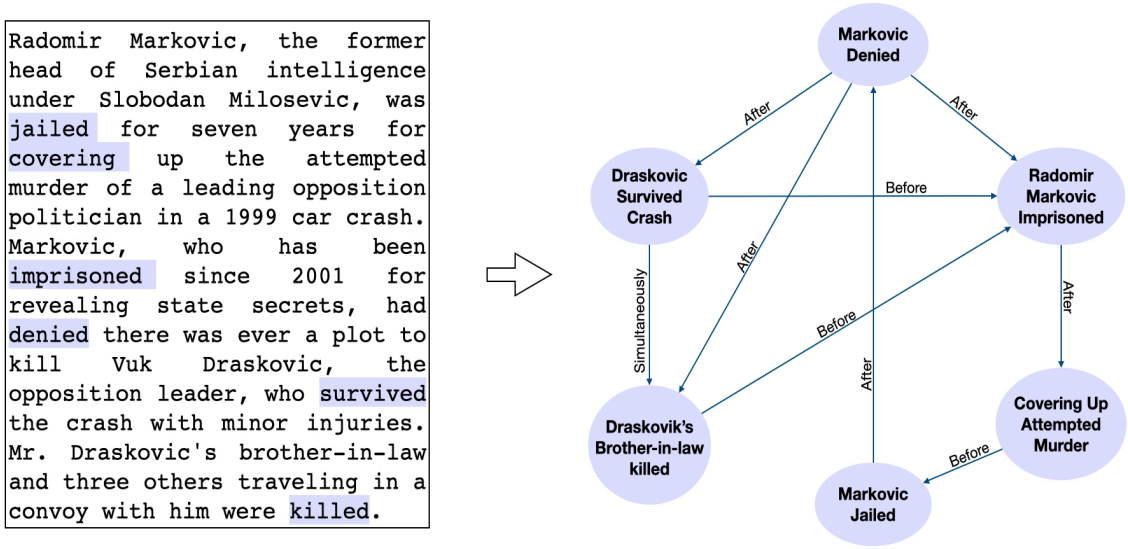


Figure 2.1: Task overview: given a document (left), automatically extract a temporal graph (right).

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 paper 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, 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 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.

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.

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).

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

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

graph representation format, we use DOT as it supports a wide variety of graphs and allows 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. [2019]). 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.

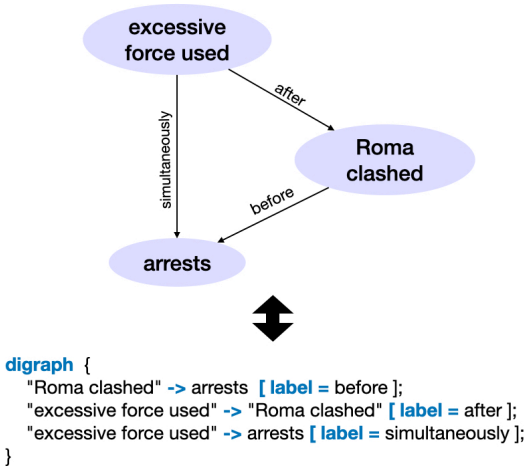


Figure 2.2: Temporal graph and the corresponding DOT representation for the sentence: *Roma clashed fiercely with the police, leading to arrests in which Roma activists said excessive force was used.*

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 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

[2] $\|$ denotes concatenation

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.

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 $p_\theta\left(\S u_i\right)$ to formulate a *masked* language modeling loss $\mathcal{L}$, which is minimized over the training corpus $\mathbf{X}$ to estimate the optimal θ:

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

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_θ 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 (**??**).

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. [2015] 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 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.

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

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	$\|\mathbf{V}\|$	$\|\mathbf{E}\|$	$d(\mathbf{V})$	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, 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.

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

Top 10 Verbs: found, killed, began, called, want, took, came, used, trying, asked **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).

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).

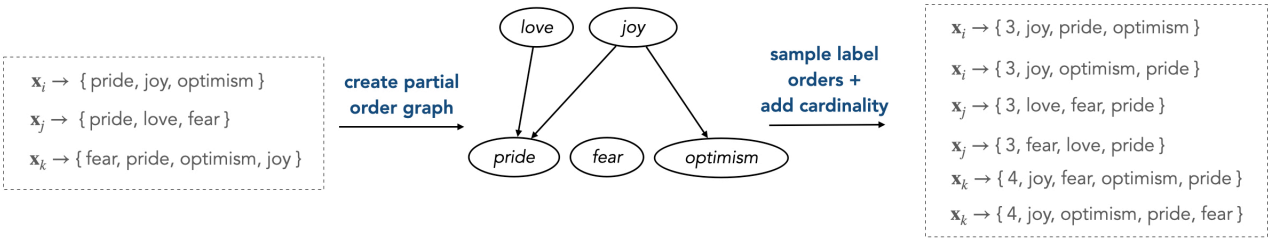


Figure 3.1: An illustrative task where given an input $\boldsymbol{x}$, the output is a set of emotions. Our method first discovers a partial order graph (middle) in which specific labels (joy) come before more general labels (pride). Listing the specific labels first gives the model more clues about 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., 2021b].

Figure **??** 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 ($\mathbb{Y}$). To see why certain orders might be more meaningful, 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 iced 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 modeling 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 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 **??** presents TSAMPLE, a novel method to create informative orders over sets tractably. Section **??** presents our method for jointly modeling cardinality and set output.

3.3.1 TSAMPLE: Adding informative orders for set output

As discussed in Section **??**, SEQ2SEQ formulation requires the output to be in a sequence. Prior work [Vinyals et al., 2016, Rezatofighi et al., 2018, Chen et al., 2021d] 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 $[\mathrm{y}_i, \mathrm{y}_j]$ (y_j should come after 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 y_i, increases the probability of 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 **??**). 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 **??**), 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 **??**). 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 **??**). 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 **??**, also shown empirically in Appendix **??**). 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: α, β
Output: n topological sorts over $G_t(V, E)$

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

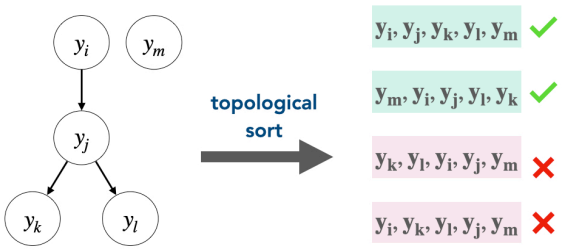


Figure 3.2: Our sampling method TSAMPLE first builds a graph G_t over the set $\mathbb{Y}_t$, and then samples orders from G_t using topological sort (`topo_sort`). The topological sorting rejects samples that do not follow the conditional probability constraints.

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

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 θ are learned (Appendix **??**).

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., 2021d]. Our formulation is closest in spirit to Chen et al. [2021d]. 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 **??**. 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 **??** in Appendix **??** 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].

TSAMPLE can apply to *any* SEQ2SEQ model. We show results with models of various capacity:

iii) BART-base [Lewis et al., 2020a] (110M),
iv) T5 [Raffel et al., 2020b] (11B), and
v) GPT-3 [Brown et al., 2020b] (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 α, β 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 **??**. We use greedy sampling for all experiments. Our method remains effective across five different sampling techniques, incl. beam search, nucleus, top-k, and random sampling (Table **??**, Appendix **??**).

[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 ∼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.

3.4.2 TSAMPLE improves existing models

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

Multi-label classification

Table 3.2 shows improvements across all datasets and models for the multi-label classification task (∼20% relative gains). For brevity, we list macro F score, and include detailed results including macro/micro precision, recall, F scores in Table **??** (Appendix **??**). 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 **??** for a detailed error analysis). We see similar trends with GPT-3 (Section **??**).

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

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

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.

Simulations

Following prior work on studying deep network properties effectively via simulation [Vinyals et al., 2016, Khandelwal et al., 2018], we design a simulation to study the effects of output order and cardinality on conditional set generation. The simulation reveals several key properties of our methods. We defer the details to Appendix **??**, and mention some key findings here. We find similar trends in simulated settings. Specifically, our method is (i) ineffective when labels are independent, (ii) helpful even when position embeddings are disabled, and (iii) helps across a wide range of sampling types.

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.

3.4.3 Why does TSAMPLE improve performance?

As mentioned in Section 6.2, 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.

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

[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.

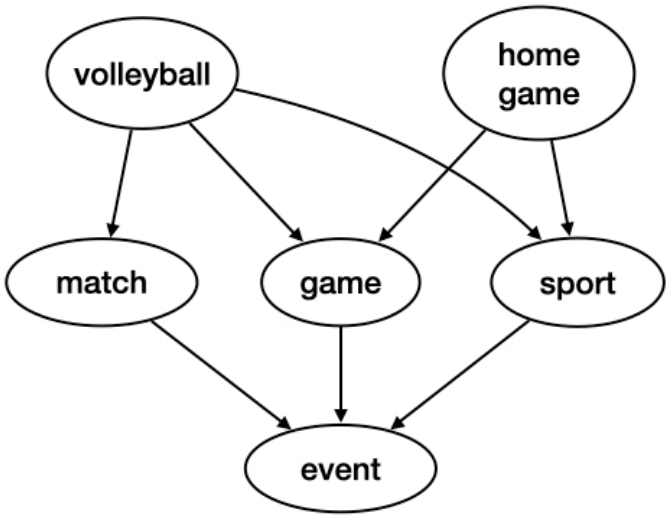


Figure 3.3: Label dependency discovered by TSAMPLE for OPENENT: specific entities (e.g., volleyball) precede generic ones (event). Appendix **??** has more examples

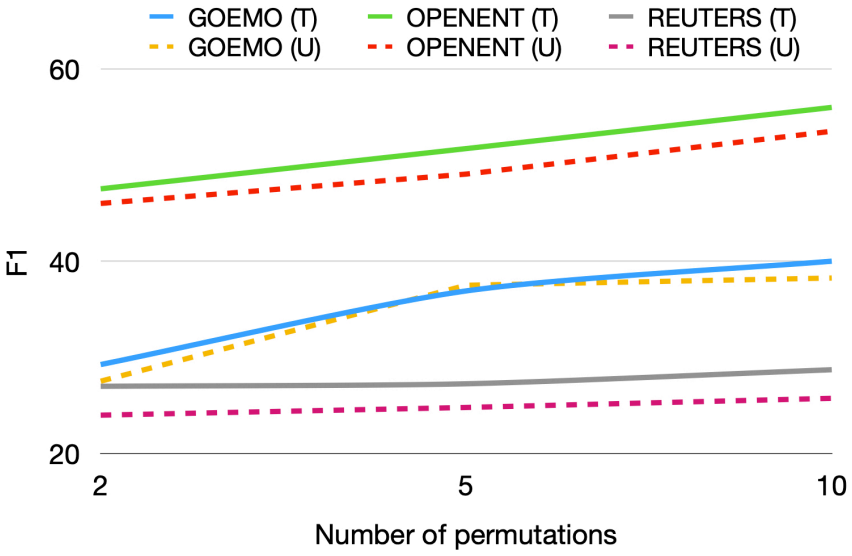


Figure 3.4: TSAMPLE (T) consistently outperforms RANDOM (U) as the number of permutations (n) is increased.

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 **??**).

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 ± 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.

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 4

Politeness Transfer: A Tag and Generate Approach

This paper introduces a new task of politeness transfer which involves converting non-polite sentences to polite sentences while preserving the meaning. We also provide a dataset of more than 1.39 million instances automatically labeled for politeness to encourage benchmark evaluations on this new task. We design a *tag* and *generate* pipeline that identifies stylistic attributes and subsequently generates a sentence in the target style while preserving most of the source content. For politeness as well as five other transfer tasks, our model outperforms the state-of-the-art methods on automatic metrics for content preservation, with a comparable or better performance on style transfer accuracy. Additionally, our model surpasses existing methods on human evaluations for grammaticality, meaning preservation and transfer accuracy across all the six style transfer tasks.

4.1 Introduction

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 the words or phrases which belong to the original style and replaces them with a tag token. If the sentence has no style attributes, as in the case for politeness transfer, the tagger adds the tag token in positions where phrases in the target style can be inserted. The *generator* takes as input the output of the tagger and generates a sentence in the target style. Additionally, unlike previous systems, the outputs of the intermediate steps in our system are fully realized, making the whole 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(§4.4). 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 (§**??**).

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.

4.2 Tasks and Datasets

4.2.1 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. 4.1). 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. 4.1). 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 (κ) of 0.77[4] and curated a final test set of 800 sentences.

[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.

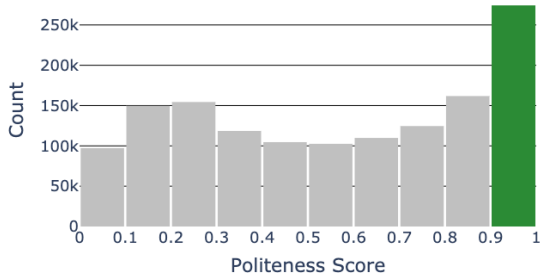


Figure 4.1: Distribution of Politeness Scores for the Enron Corpus

In Fig. 4.2, 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.

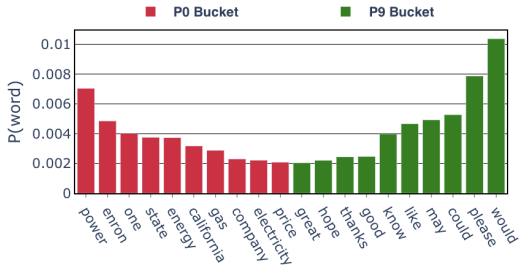


Figure 4.2: Probability of occurrence for 10 of the most common 30 words in the P_0 and P_9 data buckets

4.2.2 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].

Figure 4.3: 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 $\mathcal{S}_2$.

4.3 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 (Γ_v) which characterize the style $\mathcal{S}_v$. The presence of phrases from Γ_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 Γ_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 §4.3.2, §4.3.3.

Fig. 4.3 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.

4.3.1 Estimating Style Phrases

Drawing from Li et al. [2018], we propose a simple approach based on n-gram tf-idfs to estimate the set Γ_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. 4.2). 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 $\mathcal{S}_2$.

$$\eta_1^2(w) = \frac{\frac{1}{m}\sum_{i=1}^{m} \text{tf-idf}(w, \mathbf{x}_i^{(2)})}{\frac{1}{n}\sum_{j=1}^{n} \text{tf-idf}(w, \mathbf{x}_j^{(1)})} \tag{4.1}$$

$$p_1^2(w) = \frac{\eta_1^2(w)^\gamma}{\sum_{w'} \eta_1^2(w')^\gamma} \tag{4.2}$$

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 Γ_2 by

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

In other words, Γ_2 consists of the set of phrases in $\mathbf{X}_2$ above a given style impact k. Γ_1 is computed similarly where we use $p_2^1(w), \eta_2^1(w)$.

4.3.2 Style Invariant Tagged Sentence

The tagger model (with parameters θ_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 Γ_2/Γ_1 (§4.3.1) 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. 4.3.

$$\mathcal{L}_r(\theta_t) = -\sum_{i=1}^{|\mathbf{X}_1|} \log P_{\theta_t}(z(\mathbf{x}_i)|\mathbf{x}_i^{(1)}; \theta_t) \tag{4.3}$$

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.

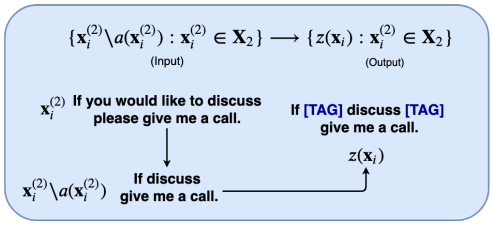


Figure 4.4: Creation of training data for add-tagger.

The training data (Fig. 4.4) for the add-tagger is given by pairs where the input is $\{\mathbf{x}_i^{(2)} \setminus a(\mathbf{x}_i^{(2)}) : \mathbf{x}_i^{(2)} \in \mathbf{X}_2\}$ and the output is $\{z(\mathbf{x}_i) : \mathbf{x}_i^{(2)} \in \mathbf{X}_2\}$. Essentially, for the input we take samples $\mathbf{x}_i^{(2)}$ 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. 4.4, 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. 4.4 is crucial for tasks like politeness transfer where one of the styles is poorly defined.

$$\mathcal{L}_a(\theta_t) = -\sum_{i=1}^{|\mathbf{X}_1|} \log P_{\theta_t}(z(\mathbf{x}_i)|\mathbf{x}_i^{(2)} \setminus a(\mathbf{x}_i^{(2)}); \theta_t) \tag{4.4}$$

	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 4.1: Results on the Politeness, Gender and Political datasets.

4.3.3 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 $\mathcal{S}_v$.

$$\mathcal{L}(\theta_g) = -\sum_{i=1}^{|\mathbf{X}_v|} \log P_{\theta_g}(\mathbf{x}_i^{(v)} | z(\mathbf{x}_i); \theta_g) \tag{4.5}$$

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 Γ_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 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."

4.4 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

	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 4.2: Results on the Yelp, Amazon and Captions datasets.

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. [2015] 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 γ in Eq. 4.2 is set to 0.75. For all datasets except Yelp we use phrases with $p_1^2(w) \geq k = 0.9$ to construct Γ_2, Γ_1 (§4.3.1). 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. [2017] 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 4.1 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.

[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 4.3: Human evaluation on Politeness, Gender, Political and Yelp datasets.

In Table 4.2, 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 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.

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.

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

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 4.3 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 4.4. 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 4.5.

[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.

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.

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

Chapter 5

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

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, GPT-2, 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]

5.1 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?

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

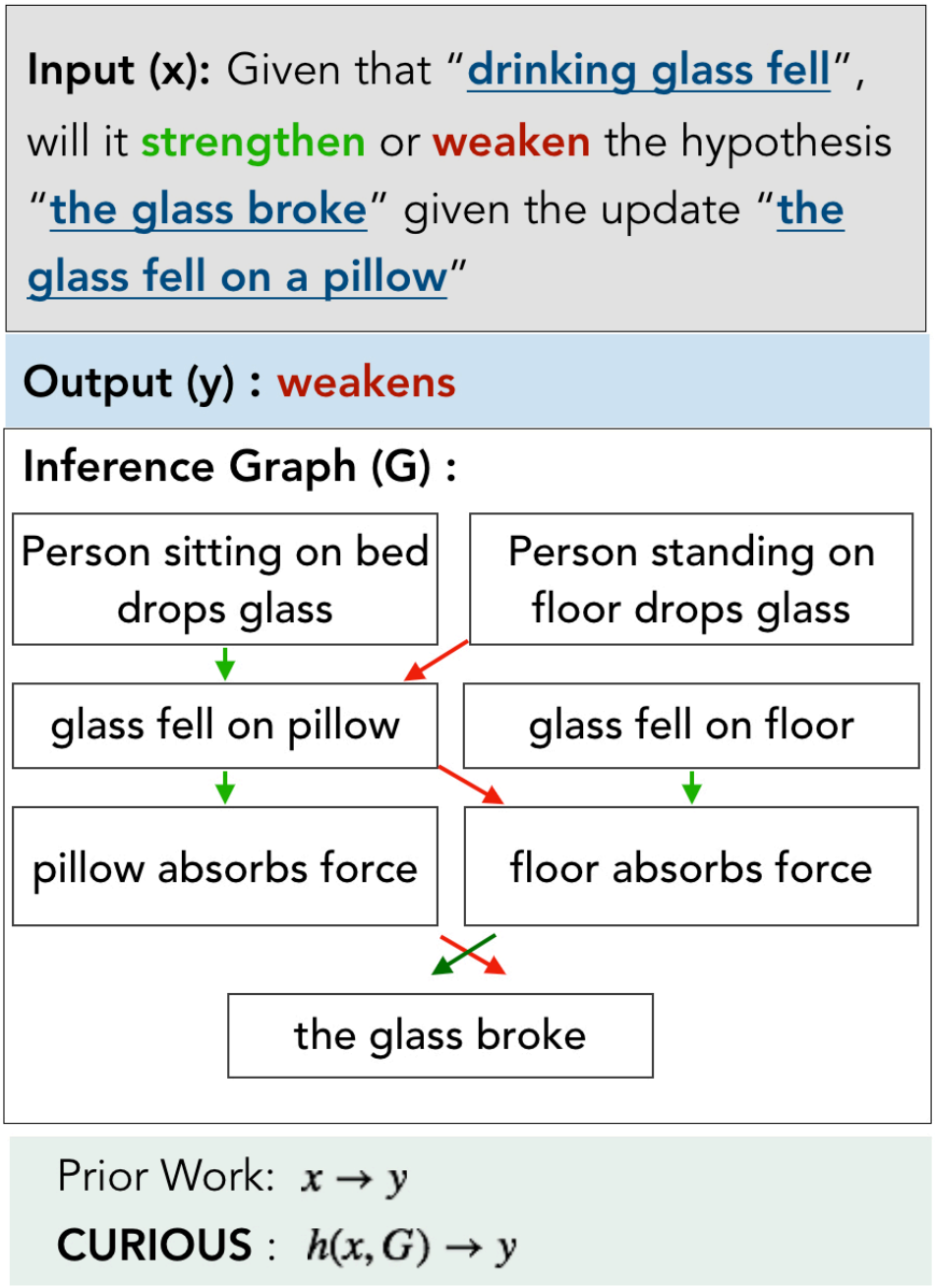


Figure 5.1: GPT-2 improves defeasible reasoning by modeling the question scenario with an inference graph G adapted from cognitive science literature. The graph is encoded judiciously using our graph encoder $h(.)$, resulting in end task performance improvement.

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, GPT-2, comprises a graph generation module and a graph encoding module to use the generated graph for the query (Figure

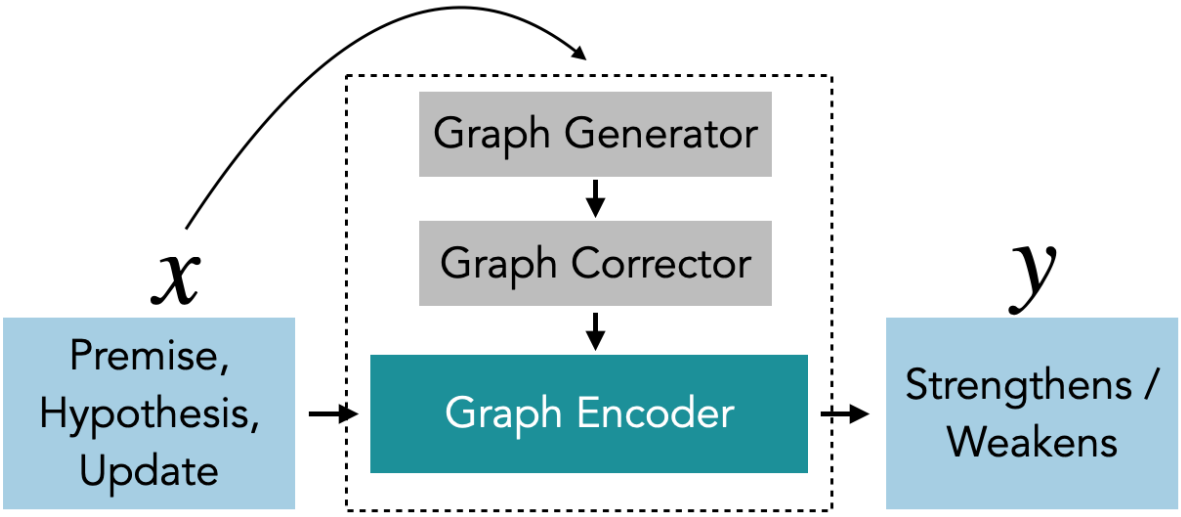


Figure 5.2: An overview of GPT-2

5.2).

To generate inference graphs, we build upon past work that uses a sequence to sequence approach Madaan et al. [2021a]. However, our analysis revealed that the graphs can often be erroneous, and GPT-2 also includes an error correction module 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., δ-SNLI (natural language inference) Bowman et al. [2015], δ-SOCIAL (reasoning about social norms) Forbes et al. [2020], and δ-ATOMIC (commonsense reasoning) Sap et al. [2019]. We show that the way the graph is encoded for input 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, GPT-2 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.

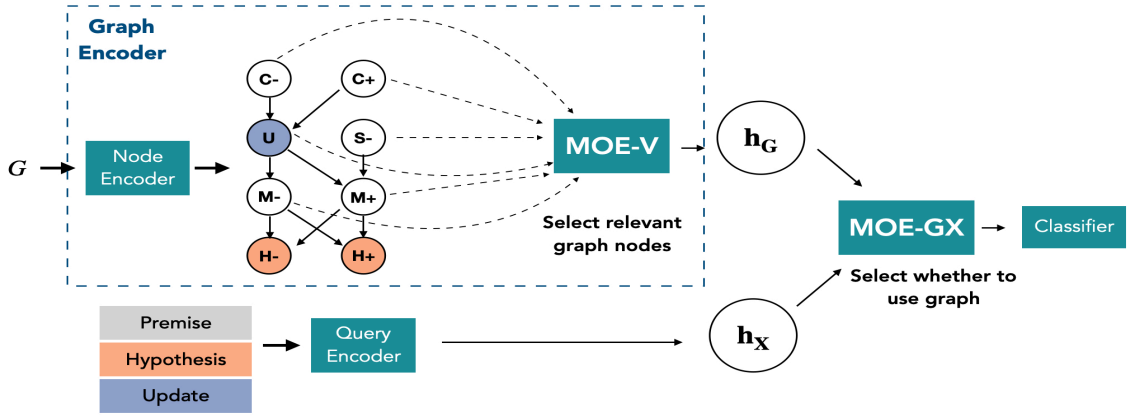


Figure 5.3: An overview of our method to perform graph-augmented defeasible reasoning using a hierarchical mixture of experts. First, **MOE-V** selectively pools the node representations to generate a representation $\mathbf{h_G}$ of the inference graph. Then, **MOE-GX** pools the query representation $\mathbf{h_x}$ and the graph representation generated by **MOE-V** to pass to the upstream classifier.

5.2 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 5.1.

5.3 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.

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 5.1, 5.4, and in Appendix section **??**.

5.3.1 Overview of GPT-2

Our system, GPT-2, comprises three components, (i) a graph generator GEN_{init}, (ii) a graph corrector GEN_{corr}, (iii) a graph encoder (Figure 5.1). GEN_{init} generates an inference graph from the input $\mathbf{x}$. We borrow the sequence to sequence approach of GEN_{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 GPT-2 includes an error correction module GEN_{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 5.2.

5.3.2 Graph generator

As the initial graph generator, we use the method described in Madaan et al. [2021a] (GEN_{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 GEN_{init} is trained as a seq2seq model, by setting $\texttt{input} =$ [Premise] $\mathbf{T}_i$ | [Situation] $\mathbf{S}_i$ | [Hypothesis] $\mathbf{H}_i$, and $\texttt{output} = \mathbf{G}_i$. 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.

5.3.3 Graph corrector

We found that 70% of the randomly sampled 100 graphs produced by GEN_{init} (undesirably) had *repeated* nodes (an example of repeated nodes is in Figure 5.4). Repeated nodes introduce noise because they violate the semantic structure of a graph, e.g., in Figure 5.4, 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 §5.4.3)

To repair such problems, we train a graph corrector, GEN_{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 section **??**. Because the nodes in the graph are from an open vocabulary, we then train a T5 sequence-to-sequence model Raffel et al. [2020b] 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 GEN_{init}. We then feed $\mathbf{G}'$ to GEN_{corr} to obtain an improved graph $\mathbf{G}$.

[2] We use their publicly available code and data

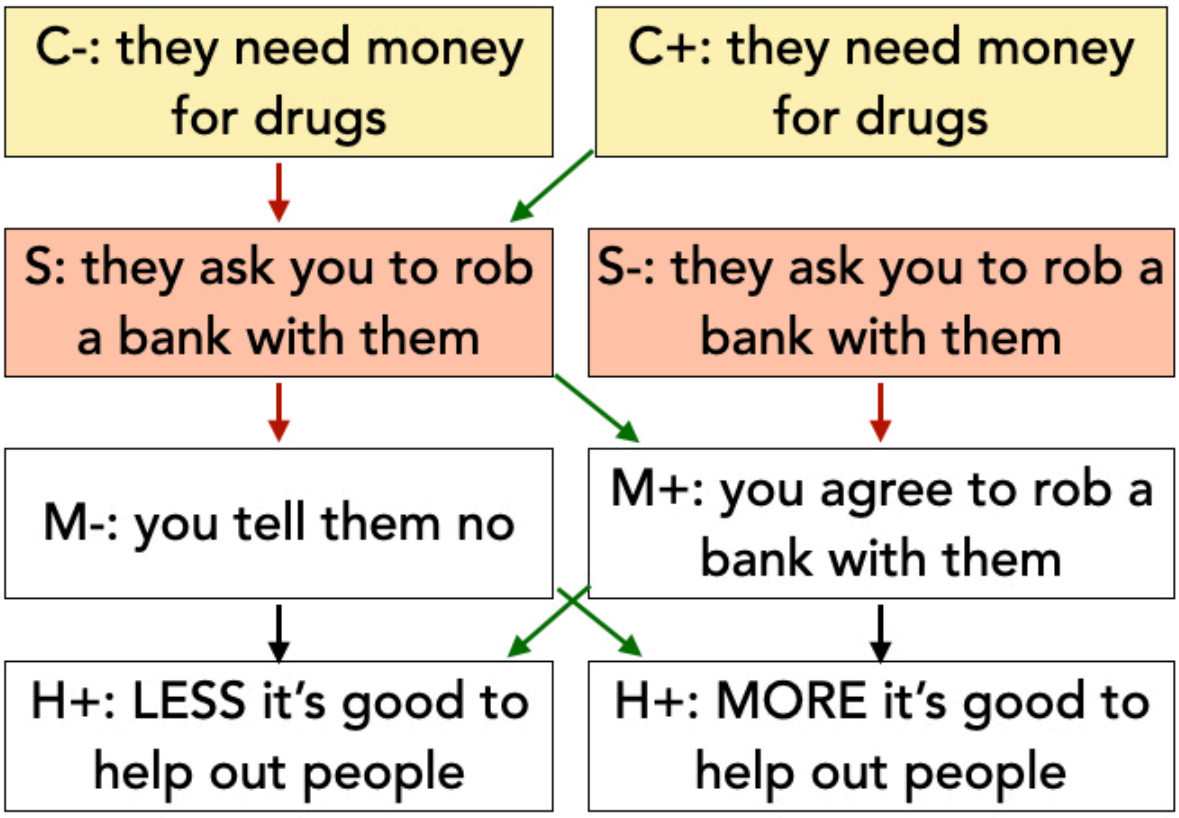


Figure 5.4: The graphs generated by GEN_{init}.The input graph has repetitions for nodes $\{C-, C+\}$ and $\{S, S-\}$. The corrected graph generated by GEN_{corr} replaces the repetitions with meaningful labels.

5.3.4 Graph Encoder

For each defeasible query (p, $\mathbf{H}$, $\mathbf{S}$), we add the inference graph $\mathbf{G}$ from GPT-2 (the corrected graph from section **??**), 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:

$$\mathbf{h_V} = [\mathbf{h}_{v_1}; \mathbf{h}_{v_2}; \ldots; \mathbf{h}_{|\mathbf{V}|}] \tag{5.1}$$

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:

$$\begin{aligned} \mathbf{p} &= \mathbf{M}(\mathbf{x}) \\ \mathbf{y} &= \sum_{i=1}^{n} p_i \mathbf{E_i}(\mathbf{x}) \end{aligned} \tag{5.2}$$

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 section **??** 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 $\mathbf{h_G}$:

$$\begin{aligned} \mathbf{p} &= \mathbf{M}(\mathbf{h_V}) \\ \mathbf{h_G} &= \sum_{v \in \mathbf{V}} p_v E_v(v) \end{aligned} \tag{5.3}$$

- **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 $\mathbf{h_x}$:

$$\begin{aligned} \mathbf{p} &= \mathbf{M}([\mathbf{h_G}; \mathbf{h_Q}]) \\ \mathbf{h_y} &= E_\mathbf{G}(\mathbf{h_G}) + E_\mathbf{Q}(\mathbf{h_Q}) \end{aligned} \tag{5.4}$$

$\mathbf{h_y}$ 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.

5.4 Experiments

In this section, we empirically investigate if GPT-2 can improve defeasible inference by first modeling the question scenario using inference graphs. We also study the reasons for any improvements.

5.4.1 Experimental setup

Datasets Our end task performance is measured on the three existing datasets for defeasible inference provided by Rudinger et al. [2020]:[3] δ-ATOMIC, δ-SNLI, δ-SOCIAL (Table 5.1). These datasets exhibit substantial diversity because of their domains: δ-SNLI (natural language inference), δ-SOCIAL (reasoning about social norms), and δ-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 GEN_{init}, and provide it to the model simply as a string (i.e., sequence of tokens; a simple, often-used

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

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

Table 5.1: Number of samples in each dataset by split.

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 section **??**. 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 section **??**.

5.4.2 Results

Table 5.2 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 δ-SNLI gaining about 4 points. GPT-2 achieves a new state-of-the-art across three datasets, as well as now producing justifications for its answers with inference graphs.

	δ-ATOMIC	δ-SNLI	δ-SOCIAL
Prev-SOTA	78.3	81.6	86.2
E2E-STR	78.8	82.2	86.7
GPT-2	**80.2***	**85.6***	**88.6***

Table 5.2: GPT-2 is better across all the datasets. This demonstrates that understanding the question scenario through generating an inference graph helps. * indicates statistical significance.

5.4.3 Understanding GPT-2 gains

In this section, we study the contribution of the main components of the GPT-2 pipeline.

Impact of graph corrector

We ablate the graph corrector module GEN_{corr} in GPT-2 by directly supplying the output from GEN_{init} to the graph encoder. Table 5.3 shows that this ablation consistently hurts across all the datasets. GEN_{corr} provides 2 points gain across datasets. This indicates that better graphs lead to better task performance, assuming that GEN_{corr} actually reduces the noise. Next, we investigate if GEN_{corr} can produce more informative graphs.

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

Table 5.3: Performance w.r.t. the graph used. $\mathbf{G}'$ is the initial graph from GEN_{init}, G is the corrected graph from GEN_{corr}. Better graphs lead to better task performance. * indicates statistical significance.

Do graphs corrected by GEN_{corr} show fewer repetitions? We evaluate the repetitions in the graphs produced by GEN_{init} and GEN_{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.

	Repetitions	GEN_{init}	GEN_{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 5.4: GEN_{corr} reduces the inconsistencies in graphs. The number of repetitions per graph and percentage of graphs with some repetition, both improve.

Table 5.4 shows GEN_{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.

Impact of graph encoder

We experiment with two alternative approaches to graph encoding to compare our MOE approach by using the graphs generated by GEN_{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 section **??**.

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 5.5 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*.

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 5.5), 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.

	δ-ATOMIC	δ-SNLI	δ-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 5.5: 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 δ-SNLI and δ-SOCIAL.

Detailed MOE analysis

We now analyze the two MoEs used in GPT-2: (i) the MOE over the nodes (**MOE-V**), and (ii) the MOE over $\mathbf{G}$ and input x (**MOE-GX**).

[4] Appendix section **??** provides an analysis on time and memory requirements.

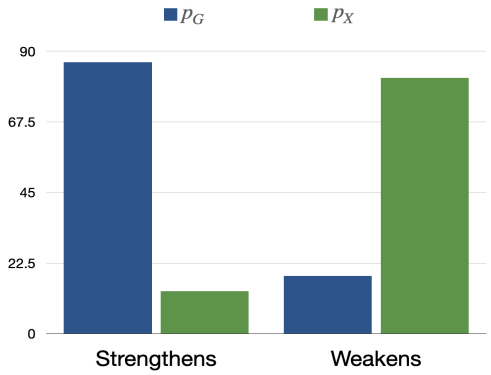


Figure 5.5: **MOE-GX** gate values for the classes strengthens and weakens, averaged over the three datasets.

MOE-GX performs better for y = strengthens: Figure 5.5 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 5.3 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 5.6 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. As our error analysis shows in §**??**, the mediators can be redundant given the query $\mathbf{x}$. Humans might have used the redundancy to reinforce their beliefs, whereas machines leverage the unique signals present in S- and the contextualizers.

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 section **??**). 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.

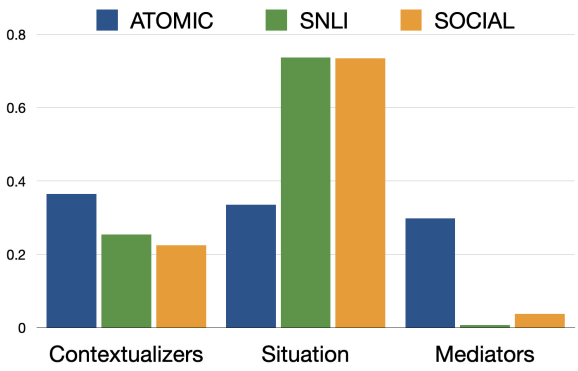


Figure 5.6: **MOE-V** gate values for the three datasets.

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 5.7 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.

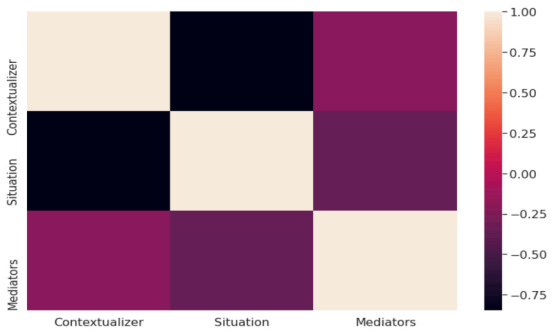


Figure 5.7: Correlation between probability assigned to each semantic type of the node by **MOE-V**

5.5 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.

Acknowledgments

We thank the anonymous reviewers for their feedback. Special thanks to reviewer 2 for their insightful comments on our MOE formulation. This material is partly based on research sponsored in part by the Air Force Research Laboratory under agreement number FA8750-19-2-0200. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

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 6 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 **??** 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 6

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.

6.1 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., 2019, Brown et al., 2020a, Chowdhery et al., 2022].

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 6.1d), or into a specification language such as DOT [Figure 6.1c; Gansner et al., 2006].

While converting the structured output into text has shown promising results Rajagopal et al. [2021], 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.

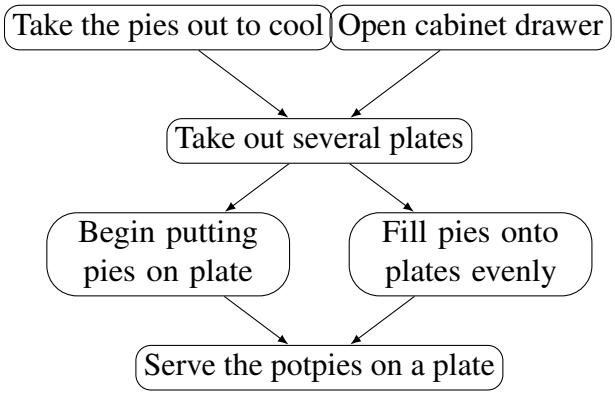
Thus, a language model which was trained on natural language text is likely to fail to capture the topology of the graph. Consequently, using LLM for graph generation typically requires a large amount of task-specific training data, and their generated outputs show structural errors and semantic inconsistencies, 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., 2021c, Xu et al., 2022a] for tasks such as code generation from natural language Austin et al. [2021a], 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 6.1.

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 $\mathcal{G}$

```python
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 6.1: An illustration of GPT-2 for the task of script generation. An input graph (6.1a) is typically represented using the DOT format (6.1c) or as a list of edges (6.1d), 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 (6.1b), allowing to model them using large-scale language models of *code*.

6.2 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 6.1) as a running example to motivate our method: script generation aims to create a script ($\mathcal{G}$) to achieve a given high-level goal ($\mathcal{T}$).

6.2.1 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 6.1b). 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 6.1b). 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 6.4.

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

We focus on large-language models of the scale of CODEX [Chen et al., 2021a]. 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. [2020a], 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 6.2 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 6.2: 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., 2021a] 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 **??** lists sample prompts for each of the tasks we experimented with.

6.3 Evaluation

We experiment with three diverse structured commonsense generation tasks: (i) script generation (PROSCRIPT, Section 6.3.2), (ii) entity state tracking (PROPARA, Section 6.3.3), and (iii) explanation graph generation (EXPLAGRAPHS, Section 6.3.4) Dataset details are included in **??**. 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.

6.3.1 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.

Baselines We experimented with the following types of baselines:

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., 2020a]. 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 6.1c). DAVINCI

[1] As of June 2022

	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 6.1: 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.

is estimated to be much larger in size than CURIE, as our experiments also reveal (Appendix **??**). DAVINCI, popularly known as GPT-3, is the strongest text-generation model available through OpenAI APIs.[2]

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 **??**.

GPT-2: We use GPT-2 to refer to setups where a CODEX is used with a Python prompt. In Section 6.4, 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.

6.3.2 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 6.1a). 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:**

[2] 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 6.2: 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.

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 6.1 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 **??**

To represent a sample for edge prediction, we list the nodes in a random order (specified after the comment `# nodes` in Figure 6.1b). 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 ($\mathrm{d}(\mathcal{G}(V))$), 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 $F_1 = \frac{2pr}{p+r}$.

Results Table 6.1 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

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 6.3: 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.

semantic metrics and structural metrics. GPT-2 outperforms DAVINCI across all semantic metrics, while DAVINCI performs slightly better in two structural metrics.

Table 6.2 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.

6.3.3 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.

We construct the Python code $\mathcal{G}_c$ as follows, and an example is shown in Figure 6.3. 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

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 6.3: 3-shots results on PROPARA. All numbers are averaged among five runs with different randomly sampled prompts. GPT-2 significantly outperforms CURIE and DAVINCI.

		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 6.4: 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.

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 6.3, 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).[3]

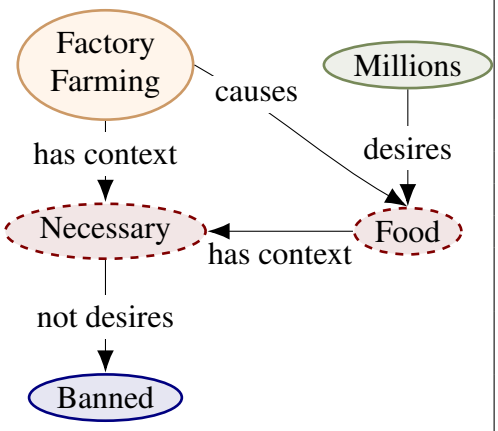
In PROPARA, GPT-2 will be ranked 6^{th} on the leaderboard.[4] However, all the methods above GPT-2 require fine-tuning on the entire training corpus. In contrast, GPT-2 uses only *3 examples* in the prompt and has a gap of less than 10 F_1 points vs. the current state-of-the-art [Ma et al., 2022]. In the few-shot settings, GPT-2 is state-of-the-art in PROPARA.

6.3.4 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*

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

[4] 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 6.4: An explanation graph (left) and the corresponding Python code (right)

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 6.4).

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 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 6.4). 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 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

	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 6.5: 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.

information, and removing any edge will hurt.

Results Table 6.4 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%.

6.4 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 Code-LLM CODEX, and a code prompt with an NL-LLM, DAVINCI. Table 6.5 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 6.3 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., 2021a, Rubin et al., 2021, Poesia et al., 2021]. We also experimented with such *dynamic* creation of the prompt, that depends on the particular

test example. Specifically, following Poesia et al. [2021], 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 6.6: 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 6.6 shows that the model was able to effectively leverage this information. In the script generation sub-task of PROSCRIPT, Table **??** shows that KST provides gains as well (Appendix **??**).

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 (**??**).

Python Formatting We performed an extensive study of the effect of the Python format on the downstream task performance in Appendix **??**. 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 (**??**) 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.

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 7 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.

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.

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 7.1: 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.

Chapter 7

MemPrompt: Memory-assisted Prompt Editing with User Feedback

7.1 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.

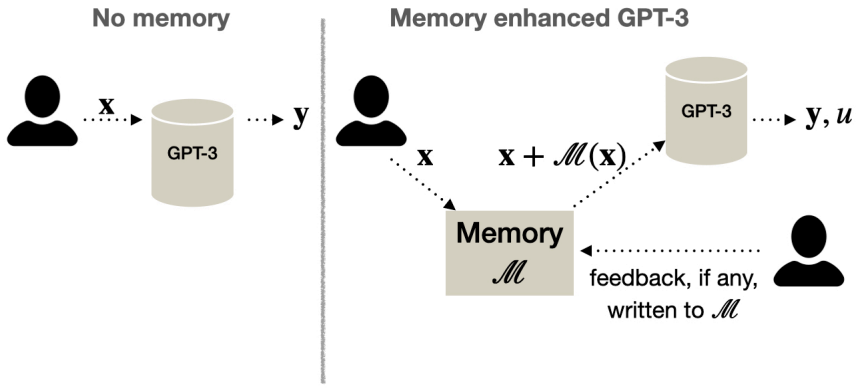


Figure 7.2: Proposed architecture: (left) GPT-3 does not account for user feedback. (right) MemPrompt maintains a memory $\mathcal{M}$ of corrective feedback, and searches for feedback from prior queries with a similar intent as x using a retrieval function $\mathcal{M}(\mathbf{x})$. x is then concatenated 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 $\mathcal{M}$.

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. [2021b] which involves editing the prompts. Our work adds interactivity to prompt engineering as it involves dynamically updating the prompt for every instance.

Figure 7.1 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.

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 imple-

mentations for each component. We then demonstrate the system on four tasks, using simulated user feedback: (1) lexical relations (e.g., antonyms, Figure 7.1), (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.

7.2 Approach

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

Table 7.1: 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 $\mathbf{p}$. 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. 7.2 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., 2020b], a prompt $\mathbf{p}$ consists of k $(\mathbf{x}, \mathbf{y})$ "in-context" examples, i.e., $\mathbf{p} = \mathbf{x}_1.\mathbf{y}_1\#\mathbf{x}_2.\mathbf{y}_2 \ldots \#\mathbf{x}_k.\mathbf{y}_k$, where $\#$ is a token separating examples and . indicates concatenation. During inference, the user inputs a question $\mathbf{x}_i$, and the model is fed $\mathbf{p} \# \mathbf{x}_i$ (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 $(\mathbf{x}, \mathbf{fb} \rightarrow \text{a}, \mathbf{y})$ in the prompt, which are aimed to teach the model to react to $\mathbf{fb}$ (Appendix **??**).

7.2.2 Verbalizing Task Understanding

Existing methods for receiving user feedback typically assume the user knows the correct answer $\mathbf{y}$ Elgohary et al. [2021]. 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. 7.3). 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 7.3: 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.

7.2.3 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 $\mathbf{x}, \mathbf{fb} \rightarrow \mathrm{a}, \mathbf{y}$ in the prompt and setting $\mathbf{fb} = \mathrm{a}$, we force the model to use the task understanding present in the input when generating the output (Figure 7.4). Recently, it has been shown that such repetition plays a crucial role in the success of few-shot prompting models [Madaan and Yazdanbakhsh, 2022].

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

7.2.4 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 section **??**). Finally, we note again that the model reacts to the feedback because some in-context samples are of the form: $(\mathbf{x}, \mathbf{fb} \rightarrow \mathrm{a}, \mathbf{y})$. We consider a diverse set of tasks $(\mathbf{x} \rightarrow \mathbf{y})$, $\mathbf{fb}$ and a, as summarized in Table 7.1.

7.2.5 Tasks

We apply our approach to four tasks: (1) lexical relations (e.g., antonyms, Figure 7.1), (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 $(\mathbf{x}, \mathbf{fb} \rightarrow \mathrm{a}, \mathbf{y})$ 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 section **??**.

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. [2020b].[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*.

7.2.6 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 $\mathbf{fb}_i$ with *clarification probability* $Pr(\mathbf{fb}_i)$. The ($\mathbf{x}_i$, $\mathbf{fb}_i$) pair is stored in a memory $\mathcal{M}$, with $\mathbf{x}_i$ as the key and $\mathbf{fb}_i$ 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

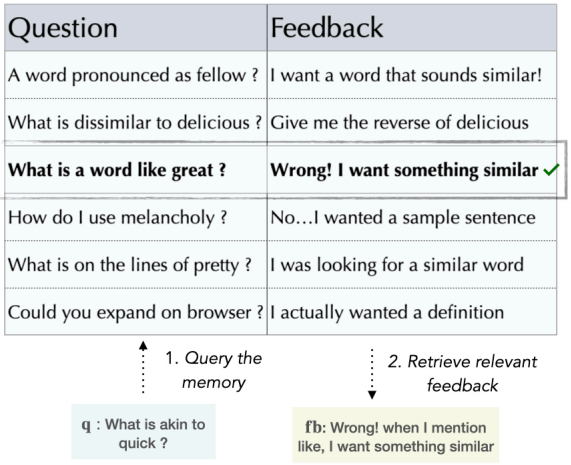


Figure 7.5: Sample snapshot of memory for lexical QA.

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 7.5 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 7.3 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 $\hat{\mathbf{fb}}$ 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 **??** 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 $\mathbf{fb}_1, \mathbf{fb}_2, \ldots, \mathbf{fb}_{|\mathcal{M}|}$ with $\hat{\mathbf{fb}}$ 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.

7.3 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 section **??**). Input is $\mathbf{p} \# \mathbf{x}_i$ (i.e., question $\mathbf{x}_i$ appended to prompt $\mathbf{p}$). It generates answer $\mathbf{y}_i$ and its understanding of the user's intent a_i.
- **GROW-PROMPT:** Similar to NO-MEM, but the $\mathbf{p}$ 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 section **??**, 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.

7.3.1 MemPrompt improves GPT-3 accuracy

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

Ethical reasoning tasks

Table 7.2 presents results on the DELPHI dataset (1,000 points in the test set). Recall from section **??** 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

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

having an 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 7.2: MemPrompt outperforms NO-MEM for both the categorical and the more challenging ERT-NL setup having longer, ambiguous inputs.

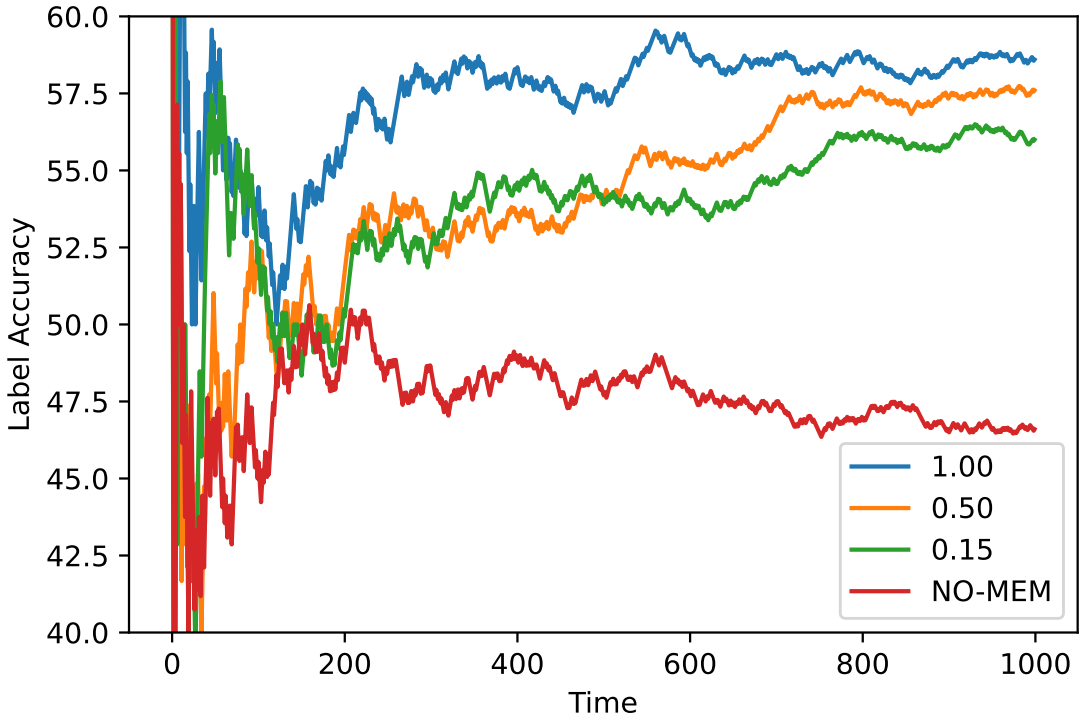


Figure 7.6: ERT-CAT: Label accuracy increases with time for all values of clarification probabilities $Pr(\mathbf{fb}_i)$.

MemPrompt effectively incorporates feedback, improving accuracy over time Figure 7.7 demonstrates that the instruction accuracy increases over time for different values of clarification probability.

Fig. 7.6 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).

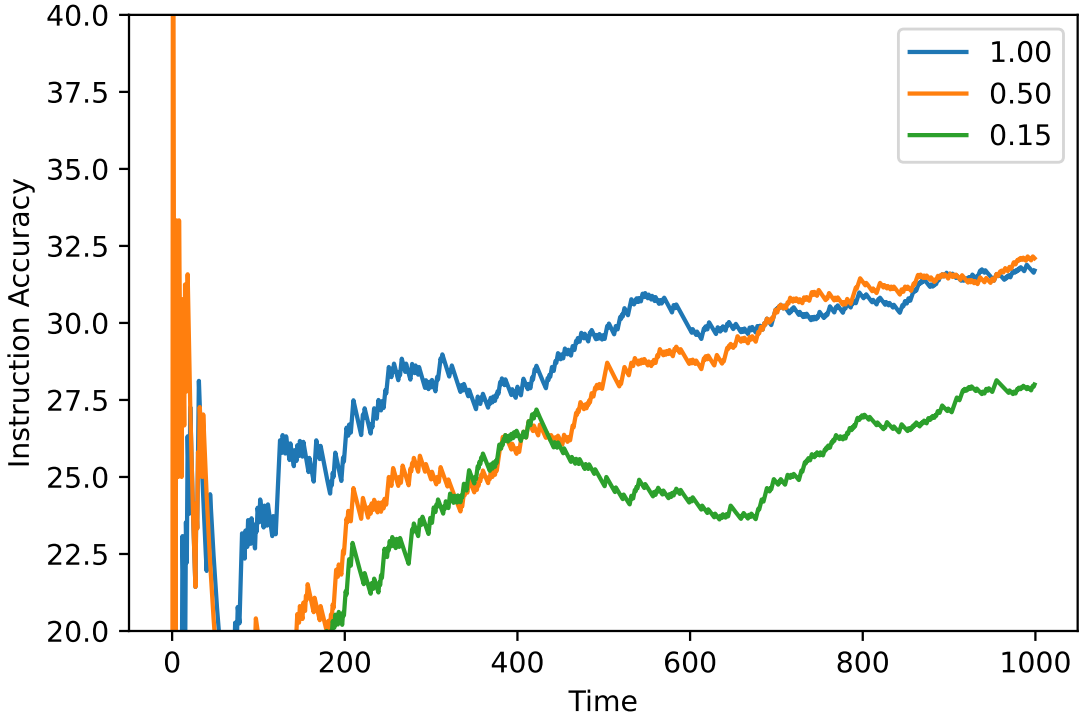


Figure 7.7: ERT-CAT: Instruction accuracy sharply increases with a larger clarification probability, showing that MemPrompt responds to feedback. With time, lower values of $Pr(\mathbf{fb}_i)$ catch up as memory is gradually filled with error cases and feedback.

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 7.2. 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 (MemPrompt)	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 7.3: ERT NL task error categories

label in these 30% cases is still correct. (Table 7.3, 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 7.3, 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 7.3, row 3). GUD-IR helps to reduce these retrieval failures. See Appendix section **??**.

Table 7.3 presents canonical examples of these error categories. We also find that over time, more relevant past examples are fetched (see Table **??**).

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 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 7.8 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 7.8, 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 7.4: Results on lexical qa: MemPrompt has the best performance across all lexical QA tasks.

7.3.2 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 section **??**).

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

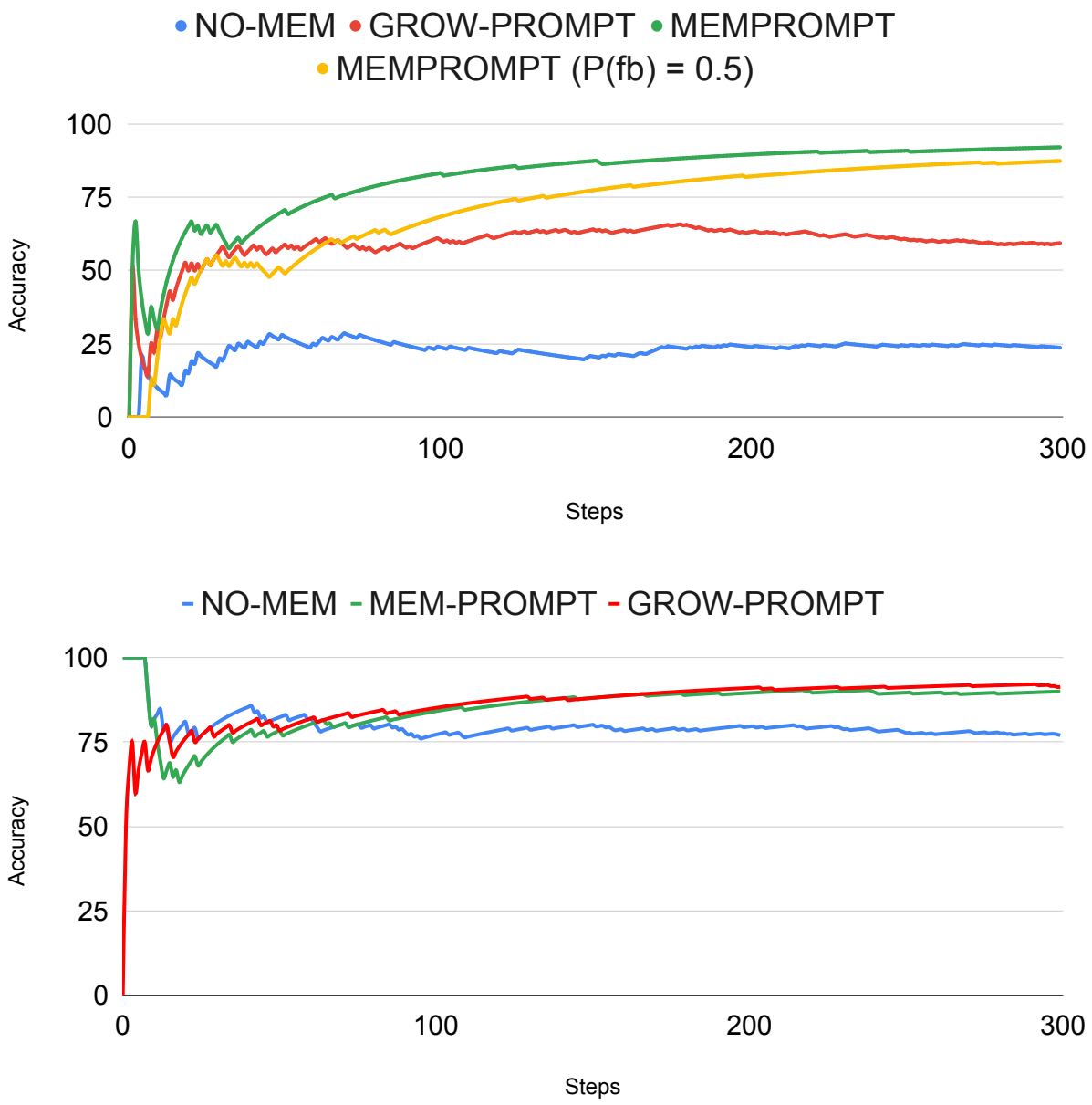


Figure 7.8: Avg. performance on lexical (top) and word scramble (bottom) tasks with time (x-axis). Accuracy increases with time as memory is filled up with feedback from past errors.

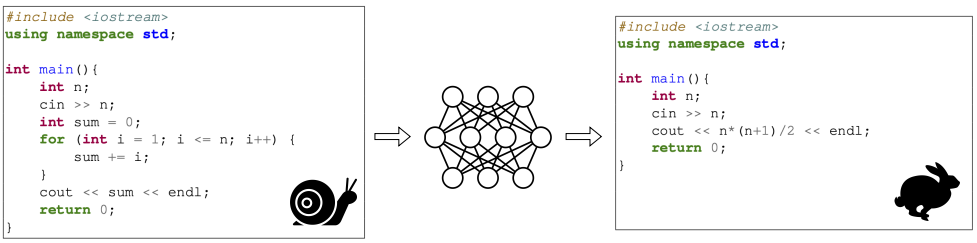
Part V

Proposed Work

While the work so far has looked at using structure at one of the phases of model development, it is natural to try and combine to include structure at more than one place. In the proposed work, we take steps towards this goal.

1. Chapter 8 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.
2. Chapter 9 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.

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.



(a) Slower Code.

(b) Faster Code.

Figure 8.1: An example of a program that solves the problem of "*compute the sum of the numbers from 1 to N*". The program in Figure 8.1(a) runs in $\Theta(N)$, whereas the program in Figure 8.1(b) runs in constant time complexity.

Chapter 8

Learning to Generate Performance Enhancing Code Edits

8.1 Introduction

Ensuring efficiency is a crucial aspect of programming, particularly when computational resources are limited or the program is utilized at a large scale.

Despite the impressive progress of optimizing compilers and other tools for performance engineering, programmers are still generally responsible for numerous performance considerations, especially in the design and implementation phase of software engineering.

Large language models (LLMs) have shown great potential in a variety of software-related tasks ranging from competitive programming [Li et al., 2021], code completion/editing [Fried et al., 2022b], to clone detection, defect detection, and much more [Lu et al., 2021, Xu et al., 2022b, Zhou et al., 2022, Austin et al., 2021b]. Motivated by these successes, we seek to answer the following question: can LLMs *improve the efficiency of programs* written in high-level programming languages like Python or C++? We hypothesize that large language models can propose rewrites that would be *impractical or non-trivial to expect from an optimizing compiler or search procedure*. If so, large language models may have an important role to play in assisting programmers in choosing more desirable refactorings.

We evaluate and improve the capacity of large language models to improve programs by curating a dataset of **P**erformance-**I**mproving **E**dits, PIE. We collect trajectories of programs written by the same user, where we track a single programmer's submission, how it evolved over time, and its performance characteristics. We demonstrate that having programs written by the same user for the same problem is important in teaching a model to propose useful performance-improving edits. Empirically, edits between these program versions are smaller than programs written by different programmers. We show that LLMs trained on these versions end up proposing smaller edits, which are more likely to preserve stylistic choices and more useful for programmers. We hypothesize that curating the dataset in this manner may make the task of program optimization easier to learn for a LLMs. We show that PIE allows us to improve the efficacy of large language models for code optimization. Specifically, we investigate the effects of using PIE for few-shot prompting on CODEX and fine-tuning models like CODEGEN. We observe significant improvements in all experiments using PIE. Ultimately, for **Python** and **C++**, these models trained on PIE can successfully optimize many examples (up to **38.3**% **and 27.9%** respectively) from the test set with substantial speedups (up to **22.64**× and **112.14**× respectively). We also demonstrate that CODEGEN-2B and CODEGEN-16B trained on PIE approaches CODEX while being up to **10**× smaller than CODEX.

8.1.1 Motivating Example

The example in Figure 8.1 demonstrates the potential of large language models for program optimization. The program in Figure 8.1(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 8.1(a) along with a comment: *// Optimize the above program.* we received the program in Figure 8.1(b) as an output. When run with an input of 100,000, the program in Figure 8.1(b) runs over 100x faster than the program in Figure 8.1(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.

8.2 The PIE Dataset

We introduce PIE, a performance-oriented dataset across multiple programming languages. Importantly, PIE captures the progressive changes a programmer makes over time to improve their code. All programs in our dataset come with unit tests that can be used to both access functional accuracy and as well as real, measured runtime. We create our dataset by branching from the CodeNet [Puri et al., 2021] collection of code samples.[1] CodeNet contains submissions

[1] https://github.com/IBM/Project_CodeNet

to competitive programming problems from two websites: AIZU and ATCODER. It also contains additional metadata such as program execution time and if submissions successfully pass test cases.

The PIE dataset is based on the insight that iterative edits made by the same developer are likely to reflect improvement while retaining the developer's stylistic choices (e.g., identifier names and program structure). We expect a model trained on such examples to produce rewrites that preserve style similar to the input program, which is more helpful in practical settings. The learning task may potentially also be easier when using program pairs from the same developer, because the difference between the input and the output directly correlates with the faster runtime. We explore these hypotheses further in **??** and **??**.

Our dataset and experiments are also unique in that we execute our programs to measure functional accuracy *as well as latency*. To our knowledge, ours is the first work involving high-level programming languages to rely on such a degree of benchmarking. Other works either rely on manual human review [Garg et al., 2022], which is hard to scale, or use proxies of latency for low-level [Shypula et al., 2021] or domain-specific [Shi and Zhang, 2020] programming languages.

8.2.1 Dataset Construction

For a given problem statement, programmers typically write an initial solution and improve it iteratively. We treat these iterative improvements as multiple versions of the program. Let $\mathbb{Y}^u = [y_1^u, y_2^u, ...]$ be a chronologically sorted series of programs, written by the user u for a problem statement x (e.g., *calculate the sum of n numbers*). These versions can differ in terms of correctness or time efficiency (e.g., y_{i+1} may pass more unit tests than y_i, or could be faster), but typically subsequent versions tend to improve performance characteristics.

Crucially, the differences between subsequent versions are typically minor — programmers usually change only small portions that affect correctness or efficiency, leaving the overall structure intact. From the series of submissions $\mathbb{Y}^u$, we remove programs that were not "*Accepted*" by the automated system; this eliminates all programs that are either incorrect (fail one or more unit tests) or take more than the allocated time to run. We then sort the set of remaining programs chronologically, yielding a *trajectory* of programs $\mathbb{Y}^* = [y_1^*, y_2^*, \ldots, y_n^*]$. Each program $y_i^* \in \mathbb{Y}^*$ is "*Correct*" in the sense that it passes all unit tests and runs within the allocated time limit. While there are multiple ways of harnessing the trajectory for a variety of problems, our focus in this work is learning to *optimize* programs. Thus, we extract (slower, faster) pairs of programs from the given trajectories.

To this end, we first divide a trajectory into pairs of the form (y_i, y_{i+1}); that is: $\mathbb{P} = (y_1, y_2), (y_2, y_3), \ldots, (y_{n-1}, y_n)$. Then, we apply a series of filtering steps on $\mathbb{P}$. Specifically, we keep pairs where the number of non-empty lines is lower than 150, and the number of different lines is less than 50%. The choice of 150 lines is motivated by the need to ensure that the programs fit within the context window of the models, which is critical for their effective processing and understanding. The 50% threshold for the number of different lines is selected to ensure that we include programs with minimal edits, thus focusing on optimization improvements that can be achieved with relatively small changes to the code.

Next, we keep pairs (y_i, y_{i+1}) for which the relative time improvement is more than 10%, that

is: $\frac{(\texttt{time}(y_i) - \texttt{time}(y_{i+1}))}{\texttt{time}(y_i)} > 10\%$ where $\texttt{time}\,(y)$ denotes the runtime of the program y. We then apply some language-specific filters to Python, such as removing unused libraries that may affect the runtime unexpectedly. Specifically, we found that certain pairs (y_i, y_{i+1}) for Python show speedup because of uninteresting reasons, such as unused imports in y_i that are removed in y_{i+1}. We use Autoflake[2] to detect and remove such instances. Finally, we compile the filtered set of pairs $\mathbb{P}$ for all unique pairs of (u, x) in our dataset, and split them to train/test/validation subsets, while ensuring that solutions for any particular problem appear in only one of the splits. **??** shows statistics of the resulting dataset, and we provide additional metadata in **??**.

Test Cases. The evaluation of generated programs is a crucial step in our experiment. Our objective is not only to generate optimized programs, but also to generate functionally correct programs. To evaluate the correctness of the generated program, we run a set of unit tests to assess its functional correctness. If a single test case fails, we reject the generated program. Ensuring the correctness of outputs is crucial for faithfully evaluating the optimization abilities of LLMs and preventing them from exploiting a lack of coverage in test cases.

CodeNet includes an average of 4 test cases per problem. We include additional test cases from AlphaCode [Li et al., 2021] generated with a fine-tuned language model to improve coverage. This additional augmentation results in 88 test cases per problem, on average. Qualitatively, we found that using the generated test cases reduces the risk of incorrect false positives – programs that are considered as functionally correct, even though they are not. In some cases, these generated test cases also helped stress-test latency better (e.g., for some inefficient programs, if the size of input integers or arrays were small enough, the differences with a generated or human reference program would be less pronounced than compared to larger inputs that came from these additional test cases).

The authors in CodeNet [Puri et al., 2021] claim the provided test case suites can be used as an oracle for determining program correctness. By using the generated test cases, we introduce even greater rigor. Nevertheless, these measures still *do not guarantee* correctness, and we believe investigating and potentially improving test case suites for PIE is an important direction for future work.

Benchmarking. While constructing the PIE dataset, we spent a large amount of time focusing on a methodology to obtain measurements of latency that would be consistent across runs. For Python, we found that we were generally able to benchmark programs using standard Python modules like SUBPROCESS and TIME as well as Linux's TASKSET command for isolating execution to specific CPUs on the host machine. However, these methods were not sufficient for benchmarking programs in C++. Because programs in C++ executed substantially faster, the variance introduced by Python's benchmarking overhead became untenable. For C++, we used HYPERFINE Peter [2023][3], a toolkit written in Rust to benchmark binaries. We found that by using HYPERFINE, we were able to reduce the coefficient of variation in benchmarking by nearly an order of magnitude, leading to more consistent results.

[2] https://github.com/PyCQA/autoflake
[3] https://github.com/sharkdp/hyperfine

8.3 Learning to Improve Code Performance

We perform experiments utilizing both *compiled* and *interpreted* programming languages: C++ and Python. In a compiled language like C++, we are also interested in demonstrating improvements in runtime on top of the compiler's optimization passes (e.g., by using the –O3 optimization flag). We experimented with two broad classes of using our models with these two languages: few-shot prompting and fine-tuning.

Few-Shot Prompting. We use CODEX (`code-davinci-002`) from OpenAI for the few-shot experiments. CODEX is based on the 175B parameter model, GPT-3[4]. For all few-shot experiments, we create a prompt of the format "$\text{slow}_1 \rightarrow \text{fast}_1$ —— $\text{slow}_2 \rightarrow \text{fast}_2$ —— ...". A slow test program is appended to this prompt during inference and supplied to the few-shot model. We create the prompts by randomly sampling from the training set, and do not perform any prompt-engineering. The prompts are shown in **??** in the Supplementary Material.

Fine-Tuning. We employ the CODEGEN [Nijkamp et al., 2022b] series of varying sizes for our fine-tuning experiments. While large language models (LLMs) such as CODEX [Chen et al., 2021b] and tools like ChatGPT can easily perform a wide range of tasks, they are not open-sourced. As a result, its accessibility and deeper study requiring access to parameters (e.g., for fine-tuning) may be limited. In contrast, CODEGEN is completely open-sourced and publicly available[5], which allows us to perform fine-tuning on the examples in PIE. CODEGEN is available for both Python (the `mono` version) and C++ (the `multi` version). CODEGEN-multi was trained on a multilingual code dataset derived from BigQuery. The `mono` variant was initialized with the `multi`, and fine-tuned on an additional 71.7B tokens of Python (See Section 2.1 of Nijkamp et al. [2022b]). We refer to CODEX and CODEGEN prompted and fine-tuned using PIE dataset as PIE-Few-shot and PIE-2B/16B. For example, for Python , we refer to CODEGEN-16B-mono fine-tuned on PIE as PIE-16B-mono. We train separate models for C++ and Python . We drop the mono and multi suffixes from model names when the usage is clear from the context.

Output Sampling. Code generation can benefit from generating multiple samples for each input [Li et al., 2021], with the best output selected according to a given metric. For example, generating multiple different outputs and choosing the one that passes the most tests or runs the fastest. In our experiments, we tested two approaches for generating output: *greedy search* and *sampling*. Greedy search generates a program by choosing the single most likely token at each step. In contrast, sampling generates a program by sampling the next token according to the LLM's output probability at each generation step. To encourage the LLM to sample from the most likely next tokens while leaving room for diversity, we set a softmax temperature of 0.7 in our experiments.[6]

After generating candidate outputs, we evaluated each for correctness using unit tests and selected the fastest output among those that passed all unit tests. We refer to this evaluation method as BEST@k, where k denotes the number of generated samples. This approach enabled us to evaluate the effectiveness of different methods for generating code and select the most optimal output among a set of candidates, based on both speed and correctness.

[4] https://platform.openai.com/docs/model-index-for-researchers
[5] https://huggingface.co/Salesforce/codegen-16B-mono
[6] https://platform.openai.com/docs/api-reference/completions

	Python			C++ (O3)		
Method	%OPT	Avg. SPEEDUP (×)	Max SPEEDUP (×)	%OPT	Avg. SPEEDUP (×)	Max S
HUMAN-REF	38.2	9.03	20.76	27.6	3.83	
CODEX	5.1	2.29	5.56	5.3	**3.99**	
CODEGEN	1.1	1.53	4.34	0.3	1.37	
SCALENE	1.4	1.09	1.13	0.7	1.27	
PIE-2B	4.4	8.61	18.55	0.5	1.11	
PIE-16B	4.4	**10.12**	**19.63**	1.5	3.03	
PIE-Few-shot	**13.1**	7.55	14.56	**8.0**	1.41	
SCALENE (BEST@16)	12.6	3.47	19.02	4.4	2.31	
PIE-2B (BEST@16)	21.1	9.37	**23.08**	3.8	2.84	
PIE-16B (BEST@16)	22.4	**9.49**	22.90	3.5	**3.48**	
PIE-Few-shot (BEST@16)	**35.2**	6.99	22.22	**27.3**	2.08	
SCALENE (BEST@32)	19.6	4.13	19.19	5.1	2.33	
PIE-2B (BEST@32)	26.3	9.25	**23.08**	5.8	2.40	
PIE-16B (BEST@32)	26.6	**9.59**	22.90	5.2	**2.81**	
PIE-Few-shot (BEST@32)	**38.3**	7.06	22.64	**27.9**	2.08	

Table 8.1: Main Results: The top block presents the performance of the human reference programs. The middle block displays the performance of CODEX, CODEGEN, SCALENE, and the PIE variants (2B, 16B, and Few-shot) using greedy decoding. The bottom two blocks show the performance of SCALENE and the PIE variants when drawing multiple samples (16 and 32), and considering the best results.

8.4 Evaluation

We evaluate the capability of LLMs to improve program runtime in two settings, (1) zero-shot and (2) fine-tuning. For the zero-shot setting, we use Codex [Chen et al., 2021b] and multiple variants of CodeGen [Nijkamp et al., 2022b], from 2B to 16B parameters.

8.4.1 Experimental Setup

Training Details

We used the pretrained checkpoints of Nijkamp et al. [2022b] that are available on the HuggingFace [Wolf et al., 2020] hub: `Salesforce/codegen-{2B,16B}-{mono,multi}`, using the `mono` models for Python and `multi` for C++ . We further trained these models using PyTorch [Paszke et al., 2017b], PyTorch Lightning [Falcon and The PyTorch Lightning team, 2019], and we used DeepSpeed [Rasley et al., 2020] to parallelize training across 4 × NVIDIA A6000 GPUs, using the `deepspeed_stage_2_offload` strategy. We mainly adopted the hyperparameters specified by Nijkamp et al. [2022b] for each model: we used a learning rate of 2e-6 for the 16B models and 5e-6 for the 2B models, a warmup proportion of 0.05, and an input/output max length of 300 tokens. In addition, we used a batch size of 1 for the 16B models with a gradient

accumulation of 2 steps, and a batch size of 4 for the 2B models. We trained each model for 5000 steps, which took about three days for the 16B models and one day for the 2B models. We list additional hyperparameters in the supplementary material (**??**).

Metrics. To evaluate performance improvement, we measured the following metrics for programs that are *functionally correct* and whose *improvements are statistically significant*:

- **Percent Optimized** [%OPT]: The proportion of programs in the test set (out of 1000 unseen samples) that are improved by a certain method.
- **Speedup** [SPEEDUP]: the absolute improvement in runtime. If o and n are the "old" and "new" runtimes, then $\text{SPEEDUP}(\text{O}, \text{N}) = \left(\frac{o}{n}\right)$.

Functional Correctness and Benchmarking. We count a program as functionally correct if it passes every test case in the test case suite described in Section 8.2.1. For benchmarking, we executed each program 5 times for each testcase (i.e. if there were 100 testcases in total for a submission, that would result in 500 runs). The system used for the benchmarking has an x86_64 CPU architecture, with an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz. The machine has 80 CPUs and runs on Linux Ubuntu 20.04.1 LTS. We run each job on a single CPU (i.e., without multiprocessing).

Statistical Significance. Controlling for variance is crucial when benchmarking programs. It is extremely unlikely for even the same program to have the exact same measured latency on two separate runs. Using a plain one-sided Student's t-test is also problematic: trivial differences in average latency may become statistically significant with a large enough sample size.

To account for these potential issues, when reporting %OPT, we require that the generated program provides a speedup that is *significantly more than 1.05x*. We statistically test this by performing an unequal variance t-test where a 5% buffer must offset the difference of means. Instead of testing if the generated program's average latency is less than or equal to the original program's latency, our null hypothesis is that the generated program is $\leq 5\%$ faster. With a large enough speedup and low-enough relative variance, we reject the null hypothesis. The t-statistic and degrees of freedom are calculated using the t-test module in R's STATS package. For all experiments, programs that pass both *correctness* and the *significance test* with (p-value < 0.05) criteria are considered to be successfully optimized in our experiments.

Baselines we evaluate our proposed approach by comparing it against four strong baselines, which are briefly described below:

Scalene: Recently, Scalene, a Python profiler Berger et al. [2022], introduced an AI-based optimization feature for code performance improvement. For our experiments, we use the prompt provided in the Scalene GitHub repository[7]. Scalene employs instruction tuning Khashabi et al. [2022] to generate more efficient variants of a given program.

Human Reference: We use human-written solutions available in PIE for each input program as reference solution. Although the Human Reference solutions exhibit faster performance, they do not achieve 100% coverage. This limitation arises from our strict criteria for determining whether a solution is indeed faster.

Codex: Codex is a 175-billion parameter model specifically trained for code generation. In our evaluation, the original slow program is supplied to the model alongside the instruction

[7]Prompt can be found at https://t.ly/kXbnI

"Optimize the above program." This allows us to assess the model's optimization capabilities without additional context or examples from the PIE dataset.

Codegen: We utilize Codegen-16B-`mono` (for Python) and Codegen-16B-`multi` (for C++) without any fine-tuning. Similar to Codex, this baseline helps us understand the efficacy of the Codegen series of models for code optimization without fine-tuning on our proposed dataset.

8.5 Results

Table 8.1 shows our main results: *large language models can optimize the runtime of programs in Python and C++, even after the C++ files were compiled with the O3 optimization level* of `gcc`. PIE-Few-shot (with 32 sampled outputs) can improve the runtime of **38.3**% of the Python and **27.9**% of the C++ examples. CODEGEN models with one tenth of Codex size trained on PIE can closely match the performance of PIE-Few-shot on speedup ([SPEEDUP]).

PIE provides valuable signals for optimization. Our proposed large language models (PIE-2B, PIE-16B, and PIE-Few-shot) outperform the other methods in terms of both the percentage of optimizations achieved (%OPT) and the average and maximum speedup (SPEEDUP). This indicates the importance of fine-tuning models on a dataset that is specifically designed for software optimization tasks, as our PIE dataset is.

Comparison with human-written reference programs. We compare human reference solutions, CODEX, CODEGEN, SCALENE, and several versions of PIE model (PIE-2B, PIE-16B, and PIE-Few-shot) with various numbers of attempts (best of 16 and best of 32). The performance metrics are the percentage of optimal solutions found (%OPT) and average, minimum, and maximum speedups (SPEEDUP).

In terms of the fraction of programs optimized, PIE-Few-shot can match human-level performance when a large number of samples are drawn (e.g., 38.2% REF vs. 38.3% PIE-Few-shot (BEST@32) for Python programs). However, it is important to recognize that human experts do not typically develop 32 different programs for a single problem, as is the case with the "best of 32" scenario. Consequently, this comparison should be considered cautiously. At the same time, one advantage of LLM-based methods is their ability to generate multiple code samples, which can then be evaluated and the best solution chosen. This can be particularly useful when finding the optimal solution is critical, and it is an advantage not easily replicated by human experts.

Performance on C++ vs. Python the performance of our proposed models, PIE-2B,16B and PIE-Few-shot, varies significantly between C++ and Python. Specifically, for Python, the performance of PIE-2B,16B is much closer to that of PIE-Few-shot, based on a 175B variant of GPT-3. We attribute this strong performance of our fine-tuned models in Python to CODEGEN-mono, a specialized model fine-tuned on a large corpus of Python. These results highlight the potential of smaller models to greatly benefit from fine-tuning and close the gap in performance with models that are an order of magnitude larger. This finding is consistent with recent research by Hoffmann et al. [2022], who found that with additional training, models with fewer parameters can compete with, and even outperform, larger language models.

Trade-off between SPEEDUP and %OPT. Our evaluation reveals a trade-off between the percentage of optimizations achieved (%OPT) and the average and maximum speedup (SPEEDUP). For example, in Python, PIE-Few-shot achieves the highest %OPT of 13.1% and an average

SPEEDUP of 7.55×, while PIE-16B achieves a higher average SPEEDUP of 10.12×, but with a lower %OPT of 4.4%. Generally, methods that achieve higher %OPT tend to have lower SPEEDUP, and vice-versa. This trade-off is expected since achieving a higher %OPT may require making many small and incremental changes that may not result in large performance improvements.

Chapter 9

Self-Refine: Iterative Refinement with Self-Feedback

9.1 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. [2022], 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 $\mathcal{M}$

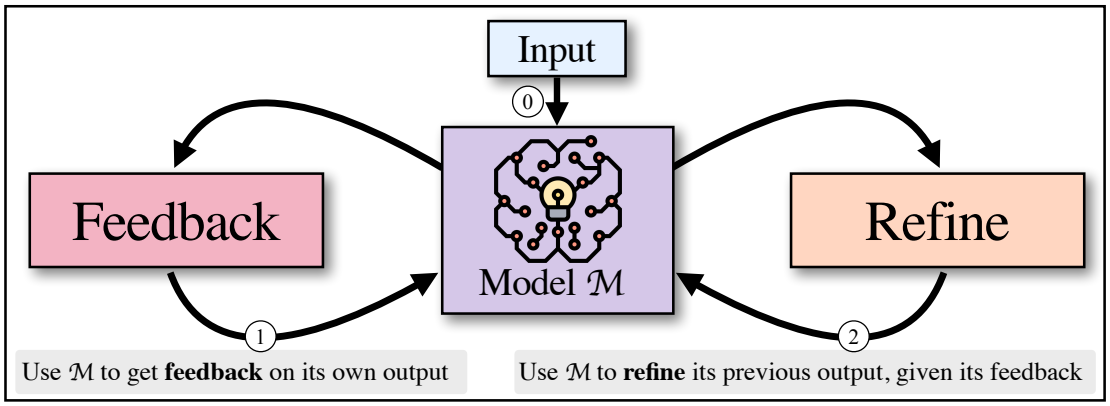


Figure 9.1: Given an input (⓪), SELF-REFINE starts by generating an output and passing it back to the same model $\mathcal{M}$ to get feedback (①). The feedback is passed back to $\mathcal{M}$, which refines the previously generated output (②). Steps (①) and (②) iterate until a stopping condition is 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., 2020a] to guide $\mathcal{M}$ to both generate feedback and incorporate the feedback into an improved draft. **??** 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., 2021b]. 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.

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. 2.4.3, 6.3.2

James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. 2.4.3

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. 9.1

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 arXiv:2108.07732*, 2021a. URL https://arxiv.org/abs/2108.07732. 6.1

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, 2021b. URL https://arxiv.org/abs/2108.07732. 8.1

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. 1

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.0473. 2.4.1

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*, 2016. 1.1.1

Jasmijn Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. Graph convolutional encoders for syntax-aware neural machine translation. *arXiv preprint arXiv:1704.04675*, 2017. 1.1.1

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 arXiv:1806.01261*, 2018. 1

Emily M. Bender and Alexander Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 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. 7.1

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. 1.1.2, 1

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. 8.4.1

Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Celikyilmaz, and Yejin Choi. COMET: Commonsense transformers for automatic knowledge graph construction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4762–4779, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1470. URL https://aclanthology.org/P19-1470. 1.1, 2.1, 2.3

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1075. URL https://aclanthology.org/D15-1075. 5.1

Penelope Brown, Stephen C Levinson, and Stephen C Levinson. *Politeness: Some universals in language usage*, volume 4. Cambridge university press, 1987. 4.1

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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 H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020a. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. 6.1, 6.2.2, 1, 9.1

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. 3.4.1, 7.2.1, 7.2.5

Paweł Budzianowski and Ivan Vulić. Hello, it's gpt-2–how can i help you? towards the use of pre-trained language models for task-oriented dialogue systems. *arXiv preprint arXiv:1907.05774*, 2019. 2.1

Yuri Burda, Roger B. Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1509.00519. 3.3.1

Meng Cao, Yue Dong, and Jackie Chi Kit Cheung. Hallucinated but factual! inspecting the factuality of hallucinations in abstractive summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3340–3354, 2022. 7.2.2

Taylor Cassidy, Bill McDowell, Nathanel Chambers, and Steven Bethard. An annotation framework for dense event ordering. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA, 2014. 2.1, 2.4.1

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, 2009. 2.1

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. 2.1, 2.2, 2.4.3

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:2107.03374 [cs]*, 2021a. URL http://arxiv.org/abs/2107.03374. arXiv: 2107.03374. 1.1, 6.2.2, 6.2.2

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. 8.3, 8.4, 9.1

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 arXiv:2107.03374*, 2021c. URL https://arxiv.org/abs/2107.03374. 6.1

Xiaohui Chen, Xu Han, Jiajing Hu, Francisco JR Ruiz, and Liping Liu. Order matters: Probabilistic modeling of node sequence for graph generation. *arXiv preprint arXiv:2106.06189*, 2021d. URL https://arxiv.org/abs/2106.06189. 3.3.1, 3.3.1

Xilun Chen, Ahmed Hassan Awadallah, Hany Hassan, Wei Wang, and Claire Cardie. Multi-source cross-lingual model transfer: Learning what to share. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3098–3112, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1299. URL https://aclanthology.org/P19-1299. 5.3.4, 5.3.4

Eunsol Choi, Omer Levy, Yejin Choi, and Luke Zettlemoyer. Ultra-fine entity typing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 87–96, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1009. URL https://aclanthology.org/P18-1009. 3.1, 3.4.1

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 arXiv:2204.02311*, 2022. URL https://arxiv.org/abs/2204.02311. 6.1

Liz Coppock. Politeness strategies in conversation closings. *unpublished paper available online at http://www. stanford. edu/ textasciitilde coppock/face. pdf (last accessed 23 December 2007)*, 2005. 4.1

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. 2.4.3, 6.3.2

Hongliang Dai, Yangqiu Song, and Haixun Wang. Ultra-fine entity typing with weak supervision from a masked language model. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 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. 3.1

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

Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 1595–1604, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1144. URL https://aclanthology.org/N18-1144. 6.3.3, 6.3.3

Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. A computational approach to politeness with application to social factors. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 250–259, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P13-1025. 4.1, 4.2.1, 6, 4.4

Dorottya Demszky, Dana Movshovitz-Attias, Jeongwoo Ko, Alan Cowen, Gaurav Nemade, and Sujith Ravi. GoEmotions: A dataset of fine-grained emotions. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 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. 3.1, 3.4.1, 3.4.1

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. 2.4.2, 4.4

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423. 1.1.2

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. 3.3.1

Rotem Dror, Gili Baumer, Segev Shlomov, and Roi Reichart. The hitchhiker's guide to testing statistical significance in natural language processing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1383–1392, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1128. URL https://aclanthology.org/P18-1128. 5.4.1

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. 2.1

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209,

San Diego, California, 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1024. URL https://aclanthology.org/N16-1024. 3.3.1

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, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.444. URL https://aclanthology.org/2021.naacl-main.444. 7.2.2

Jacqueline Evers-Vermeul, Jet Hoek, and Merel CJ Scholman. On temporality in discourse annotation: Theoretical and practical considerations. *Dialogue*

Discourse, 8(2):1–20, 2017. 2.1

William Falcon and The PyTorch Lightning team. PyTorch Lightning. *10.5281/zenodo.3828935*, 2019. URL https://github.com/Lightning-AI/lightning. 8.4.1

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021. URL https://arxiv.org/abs/2101.03961. 5.3.4, 5.3.4

Linda Flower and John R Hayes. A cognitive process theory of writing. *College composition and communication*, 32(4):365–387, 1981. 9.1

Maxwell Forbes, Jena D. Hwang, Vered Shwartz, Maarten Sap, and Yejin Choi. Social chemistry 101: Learning to reason about social and moral norms. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (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. 5.1, 2

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 arXiv:2204.05999*, 2022a. URL https://arxiv.org/abs/2204.05999. 6.1

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. 8.1

Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. A theoretical analysis of the repetition problem in text generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 12848–12856, 2021. 3.1, 3.3.2

Chuang Gan, Zhe Gan, Xiaodong He, Jianfeng Gao, and Li Deng. Stylenet: Generating attractive visual captions with styles. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3137–3146, 2017. 4.2.2

Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2006. 2.2.1, 6.1

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. 8.2

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, Jo
textasciitilde ao 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, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.gem-1.10. URL https://aclanthology.org/2021.gem-1.10. 1.1

S. Ghosh, Giedrius Burachas, Arijit Ray, and Avi Ziskind. Generating natural language explanations for visual question answering using scene graphs and visual attention. *ArXiv*, abs/1902.05715, 2019. 5.4.3

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. 1.1

Colin Graber, Ofer Meshi, and Alexander Schwing. Deep structured prediction with nonlinear output transformations. *Advances in Neural Information Processing Systems*, 31, 2018. 1.1.1

Jiatao Gu, Hany Hassan, Jacob Devlin, and Victor O.K. Li. Universal neural machine translation for extremely low resource languages. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 344–354, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1032. URL https://aclanthology.org/N18-1032. 5.3.4, 5.3.4

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. 1

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. 2.2.1

Xu Han, Tianyu Gao, Yankai Lin, Hao Peng, Yaoliang Yang, Chaojun Xiao, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. More data, more relations, more context and more openness: A review and outlook for relation extraction. *arXiv preprint arXiv:2004.03186*, 2020. 1.1.2

Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016. 4.2.2

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997. 2.4.1

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. 8.5

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rygGQyrFvH. 2.3

Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017. 1

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. 1

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, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1587–1596, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/hu17e.html. 4.3

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 *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6384–6392, 2021. 3.4.1

Daniel Im Jiwoong Im, Sungjin Ahn, Roland Memisevic, and Yoshua Bengio. Denoising criterion for variational auto-encoding framework. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 4.4

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. 5.1, 5.3.4

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 & Cognition*, 41:625–637, 2013. 7.1

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 arXiv:1907.00456*, 2019. 1.1.1

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, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-4902. URL https://www.aclweb.org/anthology/W17-4902. 4.1

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. 7.2.5, 7.3.1

Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2328–2337. PMLR, 2018. URL http://proceedings.mlr.press/v80/jin18a.html. 3.3.1

Richard A Johnson, Irwin Miller, and John E Freund. *Probability and statistics for engineers*, volume 2000. Pearson Education London, 2000. 1

P. Johnson-Laird. *Mental Models : Towards a Cognitive Science of Language*. Harvard University Press, 1983. 5.1

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994. 5.3.4

Michael I Jordan and Lei Xu. Convergence results for the em approach to mixtures of experts architectures. *Neural networks*, 8(9):1409–1431, 1995. 5.3.4

Xincheng Ju, Dong Zhang, Junhui Li, and Guodong Zhou. Transformer-based label set generation for multi-modal multi-label emotion detection. In *MM '20: The 28th ACM International Conference on Multimedia, Virtual Event / Seattle, WA, USA, October 12-16, 2020*, pages 512–520, 2020. doi: 10.1145/3394171.3413577. URL https://doi.org/10.1145/3394171.3413577. 3.1

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. 4.2.1, 4.2.1

Dániel Z Kádár and Sara Mills. *Politeness in East Asia*. Cambridge University Press, 2011. 4.1

Dongyeop Kang and Eduard Hovy. xslue: A benchmark and analysis platform for cross-style language understanding and evaluation, 2019. 4.1

Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 21:23, 2015. 2.4.3

Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. Sharp nearby, fuzzy far away: How neural language models use context. In *Proceedings of the 56th Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers), pages 284–294, Melbourne, Australia, 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1027. URL https://aclanthology.org/P18-1027. 3.4.2

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, 2022. 8.4.1

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl. 5.4.3

Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004. 4.1, 4.2.1

Robert Koons. Defeasible Reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2017 edition, 2017. 5.1

Adam R Kosiorek, Hyunjik Kim, and Danilo J Rezende. Conditional set generation with transformers. *arXiv preprint arXiv:2006.16841*, 2020. URL https://arxiv.org/abs/2006.16841. 3.1

Julia Kreutzer, Joshua Uyheng, and Stefan Riezler. Reliability and learnability of human bandit feedback for sequence-to-sequence reinforcement learning. *arXiv preprint arXiv:1805.10627*, 2018. 1.1.1

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. 1.1.1, 1.1.2

David Lewis. Reuters-21578 text categorization test collection, distribution 1.0. *http://www. research/. att. com*, 1997. 3.4.1

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 *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 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. 3.4.1

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. 7.1

Juncen Li, Robin Jia, He He, and Percy Liang. Delete, retrieve, generate: a simple approach to sentiment and style transfer. In *Proceedings of the 2018 Conference of the North American*

Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 1865–1874, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1169. URL https://www.aclweb.org/anthology/N18-1169. 4.1, 4.2.2, 4.3.1, 4.3.2, 4.4, 4.4, 4.4

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, 2021. URL https://arxiv.org/abs/2105.12655. 8.1, 8.2.1, 8.3

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. 2.1

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. 2.4.2, 4.4

Xiao Ling and Daniel S Weld. Temporal information extraction. In *AAAI*, volume 10, pages 1385–1390, 2010. 2.1

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What Makes Good In-Context Examples for GPT-3? *arXiv:2101.06804 [cs]*, 2021a. URL http://arxiv.org/abs/2101.06804. arXiv: 2101.06804. 6.4

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*, 2021b. 3.1, 7.1

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 arXiv:1907.11692*, 2019. URL https://arxiv.org/abs/1907.11692. 5.3.4, 5.4.1

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *ArXiv preprint*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664. 8.1

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1166. URL https://aclanthology.org/D15-1166. 2.4.1

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. 5.4.3

Kaixin Ma, Filip Ilievski, Jonathan Francis, Eric Nyberg, and Alessandro Oltramari. Coalescing global and local information for procedural text understanding. *arXiv preprint arXiv:2208.12848*, 2022. 6.3.3

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. 6.1

Aman Madaan and Amir Yazdanbakhsh. Text and patterns: For effective chain of thought, it takes two to tango. *arXiv preprint arXiv:2209.07686*, 2022. 7.2.3

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. 5.1, 5.3, 5.3, 5.3.1, 5.3.2, 5.3.4, 5.4.3

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. 6.1

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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6291–6310, Online and Punta Cana, Dominican Republic, November 2021c. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.508. URL https://aclanthology.org/2021.emnlp-main.508. 9.1

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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6291–6310, 2021d. 6.1

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. 2.1

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. Generation-augmented retrieval for open-domain question answering. In *Proceedings*

of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 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. 7.2.6

Gary Marcus. Experiments testing gpt-3's ability at commonsense reasoning: results., 2021. URL https://cs.nyu.edu/~davise/papers/GPT3CompleteTests.html. 7.1

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947. 5.4.1

Ardith J Meier. Defining politeness: Universality in appropriateness. *Language Sciences*, 17(4): 345–356, 1995. 4.1

Rui Meng, Sanqiang Zhao, Shuguang Han, Daqing He, Peter Brusilovsky, and Yu Chi. Deep keyphrase generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 582–592, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1054. URL https://aclanthology.org/P17-1054. 3.1, 3.4.1, 3.4.2

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 arXiv:1909.03590*, 2019. URL https://arxiv.org/abs/1909.03590. 3.1

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017. 4.4

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 arXiv:1604.01696*, 2016. URL https://arxiv.org/abs/1604.01696. 6.3.2

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. 3.1

Allen Newell, Herbert Alexander Simon, et al. *Human problem solving*, volume 104. Prentice-hall Englewood Cliffs, NJ, 1972. 1.1.2

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. 1.1, 6.1

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. 8.3, 8.4, 8.4.1

Qiang Ning, Ben Zhou, Zhili Feng, Haoruo Peng, and Dan Roth. Cogcomptime: A tool for understanding time in natural language. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 72–77, 2018. 2.1

Qiang Ning, Zhili Feng, and Dan Roth. A structured learning approach to temporal relation extraction. *arXiv preprint arXiv:1906.04943*, 2019. 2.2.1

Qiang Ning, Hao Wu, Rujun Han, Nanyun Peng, Matt Gardner, and Dan Roth. Torque: A reading comprehension dataset of temporal ordering questions. *arXiv preprint arXiv:2005.00242*, 2020. 2.1, 2.4.4

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://www.aclweb.org/anthology/Q18-1027. 4.2.1, 4.4

OpenAI. Model index for researchers. https://platform.openai.com/docs/model-index-for-researchers. Accessed: May 14, 2023. 9.1

OpenAI. Openai completion engine (davinci) api, 2021. URL https://beta.openai.com/docs/guides/completion. 3.4.1

OpenAI. Gpt-4 technical report, 2023. 9.1

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 arXiv:2110.10819*, 2021. 1.1.1

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:2203.02155, 2022a. URL https://arxiv.org/abs/2203.02155. 9.1, 9.1

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. 1.1.1

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040. 2.4.2, 4.4

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*, 2017a. 1.1

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. *NeurIPS Workshop Autodiff*, 2017b. 8.4.1

Judea Pearl et al. Models, reasoning and inference. *Cambridge, UK: CambridgeUniversityPress*, 2000. 1

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL https://aclanthology.org/D14-1162. 2.4.1

David Peter. hyperfine, 2023. URL https://github.com/sharkdp/hyperfine. 8.2.1

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, June 2011. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W11-0711. 4.1

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 *International Conference on Learning Representations*, 2021. 6.4

J. Pollock. Defeasible reasoning. *Cogn. Sci.*, 11:481–518, 1987. 5.1

J. Pollock. A recursive semantics for defeasible reasoning. In *Argumentation in Artificial Intelligence*, 2009. 5.1, 5.3

Shrimai Prabhumoye, Yulia Tsvetkov, Ruslan Salakhutdinov, and Alan W Black. Style transfer through back-translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 866–876, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1080. URL https://www.aclweb.org/anthology/P18-1080. 4.1, 4.2.2, 4.3, 4.4

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. 6.3.2

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 arXiv:2105.12655*, 2021. URL https://arxiv.org/abs/2105.12655. 8.2, 8.2.1

Kechen Qin, Cheng Li, Virgil Pavlu, and Javed Aslam. Adapting RNN sequence prediction model to multi-label set prediction. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3181–3190, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1321. URL https://aclanthology.org/N19-1321. 3.1

Alec Radford. Improving language understanding by generative pre-training. 2018. 1.1

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. 2.3

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. *arXiv preprint arXiv:1910.10683*, 2019. URL https://arxiv.org/abs/1910.10683. 6.1

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. *Journal of Machine Learning Research*, 21(140):1–67, 2020a. URL http://jmlr.org/papers/v21/20-074.html. 1.1

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. *Journal of Machine Learning Research*, 21:1–67, 2020b. 3.4.1, 5.3.3

Dheeraj Rajagopal, Aman Madaan, Niket Tandon, Yiming Yang, Shrimai Prabhumoye, Abhilasha Ravichander, Peter Clark, and Eduard Hovy. Curie: An iterative querying approach for reasoning about situations, 2021. 6.1

Sudha Rao and Joel Tetreault. Dear sir or madam, may i introduce the gyafc dataset: Corpus, benchmarks and metrics for formality style transfer. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 129–140, 2018. 4.1

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM, 2020. URL https://dl.acm.org/doi/10.1145/3394486.3406703. 8.4.1

Alexander J Ratner, Henry Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. Learning to compose domain-specific transformations for data augmentation. *Advances in neural information processing systems*, 30, 2017. 1.1.1

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, 2016. 4.2.2

Machel Reid and Graham Neubig. Learning to model editing processes. *arXiv preprint arXiv:2205.12374*, 2022. 9.1

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410. 7.2.6

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 arXiv:1805.00613*, 2018. URL https://arxiv.org/abs/1805.00613. 3.1, 3.3.1

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning To Retrieve Prompts for In-Context Learning. *arXiv:2112.08633 [cs]*, 2021. URL http://arxiv.org/abs/2112.08633. arXiv: 2112.08633. 6.4

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. 5.1, 5.2, 5.4.1, 5.4.1

Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010. 1.1.2

Swarnadeep Saha, Prateek Yadav, Lisa Bauer, and Mohit Bansal. ExplaGraphs: An Explanation Graph Generation Task for Structured Commonsense Reasoning. *arXiv:2104.07644 [cs]*, 2021. URL http://arxiv.org/abs/2104.07644. arXiv: 2104.07644. 6.1, 6.3.4, 6.3.4

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. 5.4.3

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. 6.1, 2, 6.3.2, 6.3.2, 6.3.2

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. 5.1

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. 2022. doi: 10.48550/ARXIV.2208.11663. URL https://arxiv.org/abs/2208.11663. 9.1

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015. 1.1.2

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015. 4.4

Shikhar Sharma, Layla El Asri, Hannes Schulz, and Jeremie Zumer. Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation. *CoRR*, abs/1706.09799, 2017. URL http://arxiv.org/abs/1706.09799. 3

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 *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=B1ckMDqlg. 5.3.4, 5.3.4

Tianxiao Shen, Tao Lei, Regina Barzilay, and Tommi Jaakkola. Style transfer from non-parallel text by cross-alignment. In *Advances in neural information processing systems*, pages 6830–6841, 2017. 4.1, 4.2.2, 4.3, 4.4, 4.4

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. 4.2.1

Hui Shi and Yang Zhang. Deep Symbolic Superoptimization without Human Knowledge. *ICLR 2020*, 2020. 8.2

M. Shi, Yufei Tang, Xingquan Zhu, and J. Liu. Feature-attention graph convolutional networks for noise resilient learning. *ArXiv*, abs/1912.11755, 2019. 5.4.3

Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. Learning to Superoptimize Real-world Programs. *ArXiv preprint*, abs/2109.13498, 2021. URL https://arxiv.org/abs/2109.13498. 8.2

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. 9.1

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. 4.4

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020. 1.1.1

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. 2.1

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 Con-*

ference 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. 7.1

Niket Tandon, Bhavana Dalvi, Keisuke Sakaguchi, Peter Clark, and Antoine Bosselut. WIQA: A dataset for "what if..." reasoning over procedural text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6076–6085, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1629. URL https://aclanthology.org/D19-1629. 5.3, 5.3.2, 6.1

Naushad UzZaman and James F Allen. *Interpreting the temporal aspects of language*. Citeseer, 2012. 2.4.3

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, 2013. 2.4.3

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. 1.1.2, 1, 2.3, 4.4, 5.4.3

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06391. 3.1, 3.1, 3.2.1, 3.3.1, 3.4.2

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, May 2018. European Language Resources Association (ELRA). URL https://www.aclweb.org/anthology/L18-1445. 4.2.2

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. 6.1

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified

pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. URL https://arxiv.org/abs/2109.00859. 1.1, 6.1

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. 6.3.2

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SJeYe0NtvH. 3.1, 3.3.2

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*, 2022. 9.1

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. 3.3.1

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, abs/1910.03771*, 2019. 1.1

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6. 8.4.1

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022a. URL https://arxiv.org/abs/2202.13169. 6.1, 6.2

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A Systematic Evaluation of Large Language Models of Code. In *MAPS*, 2022b. 8.1

Wei Xu, Alan Ritter, Bill Dolan, Ralph Grishman, and Colin Cherry. Paraphrasing for style. In *Proceedings of COLING 2012*, pages 2899–2914, Mumbai, India, December 2012. The COLING 2012 Organizing Committee. URL https://www.aclweb.org/anthology/C12-1177. 4.1

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. 3.1

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 *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1008–1025, 2020. 3.1

Zichao Yang, Zhiting Hu, Chris Dyer, Eric P Xing, and Taylor Berg-Kirkpatrick. Unsupervised text style transfer using language models as discriminators. In *Advances in Neural Information Processing Systems*, pages 7287–7298, 2018b. 4.3

Thomas Wolf Yangfeng Ji, Antoine Bosselut and Asli Celikyilmaz. The amazing world of generation. *EMNLP tutorials*, 2020. URL https://nlg-world.github.io/. 1.1

Jiacheng Ye, Tao Gui, Yichao Luo, Yige Xu, and Qi Zhang. One2Set: Generating diverse keyphrases as a set. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 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. 3.1, 3.1, 3.4.2, **??**

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, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5694–5703. PMLR, 2018. URL http://proceedings.mlr.press/v80/you18a.html. 1.1

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 *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 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. 3.1

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. Amr parsing as sequence-to-graph transduction. *arXiv preprint arXiv:1905.08704*, 2019a. 1.1.2

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020a. URL https://openreview.net/forum?id=SkeHuCVFDr. 6.3.4

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. 3.3.2

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 *ACL, system demonstration*, 2020b. 1.1

Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. Ernie: Enhanced language representation with informative entities. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1441–1451, 2019c. 1.1.2

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao JIang, and Graham Neubig. DocPrompting: Generating Code by Retrieving and Reading Docs. *ArXiv preprint*, abs/2207.05987, 2022. URL https://arxiv.org/abs/2207.05987. 8.1