

# A CODESPACE AUTOENCODER FOR LANGUAGE TASKS

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Modern language modeling datasets require models to handle compositional reasoning, fact recall, and task-specific constraints. While these tasks are expressed in natural language, they often imply an underlying symbolic representation. In this work, we consider methods for extracting a latent symbolic representation in an unsupervised manner. We propose an autoencoder that models observed text data as being generated from underlying code with a dataset level function library. Our method is non-parametric and leverages in-context learning and code interpretation for inference. Code as the latent symbolic representation offers two key advantages. First, code offers a structured space that can be explored via modular functions; second, code is interpretably executable using deterministic and neural interpreters, enabling compositional and programmatic decoding into text. By identifying and composing patterns in this latent space, we can sample programs that produce correct, diverse, and task-relevant text through program execution. We demonstrate how our method induces a latent space with modern LLMs, explore patterns discovered within it, and evaluate text data synthesized from our induced latent space.

## 1 INTRODUCTION

Many natural tasks expressed with language have an underlying symbolic process. These latent processes can be represented well by code. For example, consider the example in Figure 1. Correct interpretation and execution of the task requires one to identify processes useful for the task (parsing time strings, comparing timestamps), then compose them properly (compose identified relations). We can enable modeling of more complex tasks by pairing neural and symbolic processes: neural processes discover the underlying symbolic structure, and symbolic processes execute the identified representation.

Given a statement about date and time, state whether the statement is true or false. The number of date/time operands in the statement ranges between 2 and 3. Let’s say the values are denoted by t1, t2, and t3. The statements follow one of the following ten templates: ‘t1 occurs before t2, t1 doesn’t occur before t2, t1 occurs after t2, t1 doesn’t occur after t2, t1 occurs between t2 and t3, t1 doesn’t occur between t2 and t3, t1 occurred before t2 but after t3, t1 occurred after t2 but before t3, t1 didn’t occur before t2 but after t3, t1 didn’t occur after t2 but before t3’. The output should be either ‘True’ or ‘False’.  
Input: 22:01:41 occurs between 08:47:09 and 11:26:35 AM.  
Output: False

```
def is_before()      def is_between()
def is_after()      re.findall()
datetime.strptime()

def is_before(t1, t2):
    return datetime.strptime(t1, "%H:%M:%S") <
           datetime.strptime(t2, "%H:%M:%S")

def is_after(t1, t2):
    return datetime.strptime(t1, "%H:%M:%S") >
           datetime.strptime(t2, "%H:%M:%S")

def is_between(t1, t2, t3):
    return is_before(t1,t3) and
           is_after(t1,t2)

input = "22:01:41 occurs between 08:47: [...]
t1,t2,t3 = re.findall(r'\d+:\d+:\d+', input)
task_output = is_between(t1,t2,t3)
```

Figure 1: The text inquiry (Wang et al., 2022) (left) has an underlying symbolic representation represented by a task-level library of functions (top right) and an instance-level code program (right).

Autoencoders are one way to discover these symbolic representations. Autoencoders model mappings to and from a latent space for a given dataset. Exposing this underlying distribution allows us

054 to manipulate latent processes to decode into domain-relevant text with desired attributes (Bowman  
055 et al., 2016).

056 In this work, we propose an autoencoder that models observed text datasets as being generated from  
057 underlying code with a dataset-level function library. Our method is non-parametric and leverages  
058 in-context learning (ICL) (Brown, 2020) with per-dataset demonstrations to model the encoder, de-  
059 coder and prior of the latent codespace. Code as the latent representation allows us to use determin-  
060 istic programmatic and neural interpreters (Li et al., 2024) to map from latent space back into text.  
061 We sustain coverage over all task domains with code programs by interleaving compute operations,  
062 fact retrieval, and reasoning steps as necessary. Code programs are composed of modular function;  
063 we use these functions to conditionally sample from the latent space to sample and produce text  
064 according to target attributes.

065 Our method extracts the symbolic latent space underlying different datasets with minimal human  
066 effort by inducing an unsupervised non-parametric autoencoder from a set of given text examples.  
067 In this paper, we show that across a variety of datasets, our method effectively unveils underlying  
068 symbolic patterns in the datasets. Samples from the induced latent codespace produce more correct  
069 and domain-relevant text examples than do non-latent modeling approaches and latent modeling  
070 approaches without code as the latent representation. We find that leveraging an executable latent  
071 representation is particularly helpful in task settings that are compositional and algorithmic in nature.  
072 We also evaluate the effect of using our method to generate synthetic data to train a downstream  
073 model for a given task.

## 075 2 A NON-PARAMETRIC LATENT MODEL FOR DATASETS

076 Define a dataset as  $x = \{x_i = (t_i, y_i)\}^N$ , consisting of a sequence of text-output pairs. We are  
077 interested in defining a latent representation of this dataset,  $z = \{z_\ell, z_1, \dots, z_N\}$  consisting of a  
078 global term  $z_\ell$  and datapoint specific terms  $z_i$ . For each datapoint, we are interested in finding a  
079 latent symbolic representation  $z_i \in \mathcal{Z}$ . In addition, we posit that the dataset has a global shared  
080 library structure  $z_\ell$ .

082 Our goal is to infer the latent representation  $z$  from the dataset. We write this as a posterior inference  
083 problem,  $p(z|x)$ . Inference of this distribution is intractable since it requires marginalizing over all  
084 representations. Instead, optimize a variational posterior  $q$  to approximate this objective:

$$085 \arg \min_q \text{KL}(q(z|x) \parallel p(z|x)) = \text{KL}(q(z_\ell|x) \parallel p(z_\ell|x)) + \mathbb{E}_{z_\ell \sim q(z_\ell|x)} \sum_i \text{KL}(q(z_i | z_\ell, x_i) \parallel p(z_i | z_\ell, x_i))$$

$$086 = \text{KL} + \mathbb{E}_{z_\ell, z_i \sim q} \log \frac{q(z_i | z_\ell, x_i)}{p(x_i | z_i, z_\ell)p(z_i | z_\ell)}.$$

089 where the first step comes from KL identities and the assumption that each datapoint  $x_i$  is condi-  
090 tionally independent of  $z_j$  given the library  $z_\ell$ , and the second step factors out each  $x_i$ .

092 To fully specify the variational objective, we need a variational family for  $q$ . In deep learning,  
093 it is common to parameterize  $q$  and use gradient descent to minimize this equation. Instead, we  
094 use a nonparametric approach to search over a subset of concrete exemplars  $\mathcal{D}$ , as example-latent  
095 representation pairs. Specifically  $\mathcal{D}$  consists of a base library and example-symbol pairs, i.e.  $\mathcal{D} =$   
096  $\{\bar{z}_\ell, (\bar{x}_1, \bar{z}_1), \dots, (\bar{x}_M, \bar{z}_M)\}$ . Given a specific set  $\mathcal{D}$ , the variational posterior can be defined through  
097 in-context learning (ICL) (Brown, 2020). For each datapoint this is  $q(z_i|x_i, z_\ell; \text{ICL}(\mathcal{D}))$ , and the  
098 full objective is:

$$099 \arg \min_{\mathcal{D}} \text{KL}(q(z|x; \mathcal{D}) \parallel p(z|x))$$

100 We need to solve this objective by finding the best subset  $\mathcal{D}$  for the dataset. This is a combinatorial  
101 optimization problem and is intractable to solve by search. We approximate this with rejection-  
102 sampling additions to the library and new  $\tilde{z}_j$  examples. We apply an iterative optimization starting  
103 from  $\mathcal{D}^0$ . For a given dataset example  $x_j$ , we sample  $\tilde{z}_\ell$  and  $\tilde{z}_j$  from the variational posterior, as de-  
104 scribed above, and reject  $\tilde{z}$  that do not score well according to the log ratio. Library-representations  
105 pairs that score well are included in the next  $\mathcal{D}^{j+1}$ , roughly:

$$106 \mathcal{D}^{j+1} = \mathcal{D}^j \cup \{\tilde{z}_\ell, (x_j, \tilde{z}_j)\} \tag{1}$$

### 3 RELATED WORKS

**Autoencoding Datasets** Prior work has explored methods for inducing and analyzing latent variables for datasets. Bowman et al. (2016) introduce a RNN-based variational autoencoder (Kingma, 2013), which interpolates and generates coherent novel sentences by learning and sampling from a continuous posterior. In contrast, our approach operates within a structured discrete latent space that allows for direct inspection and execution. Friedman et al. (2022) present a method to diagnose datasets by inducing an underlying context-free grammar per dataset to identify “shortcuts” from which they construct adversarial examples to challenge models trained on said datasets. Other works (Qiu et al., 2024; Madaan et al., 2022) study LLMs’ ability to propose structured hypotheses over data and apply those hypotheses to new datapoints during inference. Though not explicitly framed as autoencoders, these methods share a similarity with ours, as they refine an “encoder” (structured hypothesis proposer) alongside a static “decoder” (hypothesis interpreter). However, these approaches require hypotheses to satisfy certain symbolic constraints, and therefore constrain their study to a limited set of symbolic tasks. In contrast, our work uses deterministic and neural program execution, allowing us to model a broader range of any text task.

**Library Learning and Program Induction** The advantages of breaking programming tasks into subprograms (Zelikman et al., 2022a) and inducing a shared library of reusable subprograms have been demonstrated in prior research. For example, DreamCoder (Ellis et al., 2021) is a wake-sleep algorithm for Bayesian program induction that cycles between searching for programs, building a library of common concepts in the domain, and training a neural search policy on recalled and sampled programs. Similarly, Wang et al. (2024) develop a method to induce a library of verifiable and efficient subprograms during inference. In our work, we induce a development set that optimizes our autoencoding objectives, but unlike these prior works, our approach does not require code to be fully executable. We instead use imagined functions, as in the Chain-of-Code framework (Li et al., 2024), which are emulated by a LLM.

**Intermediate Reasoning with LLMs** As LLMs become increasingly capable in language tasks, they have been used for more complex tasks requiring multi-step reasoning (Zhao et al., 2023a;b). Prior works show that prompting a model to “think aloud” during inference enhances their performance on tasks requiring reasoning (Wei et al., 2022; Nye et al., 2021). To address LLMs’ limitations in handling computational tasks or recalling updated facts, tools such as code execution, internet queries, and SMT solvers have been integrated into inference pipelines (Li et al., 2024; Chen et al., 2023; Schick et al., 2023; Press et al., 2023; Lee et al., 2024). Reasoning has also been integrated back into the model parameters: the Self-Taught Reasoner (Zelikman et al., 2022b) fine-tunes a model on its own produced and verified reasoning strings. While these works improve inference-time reasoning, they are constrained to the scope of the given data, and they do not attempt to infer structural properties of the dataset itself, which is a focus of our work.

### 4 A CODESPACE AUTOENCODER FOR LANGUAGE

To apply our framework to extracting code representation, we describe how each term described in Equation 1 can be implemented with code LLMs and interpreters. First we establish the shared structure  $z_\ell$  as the library of code functions used by latent code programs of the dataset. Each  $(x_i, z_i)$  instance of  $\mathcal{D}$  is a paired language example and corresponding code program.

The encoder is implemented as a LLM prompted by dataset-level demonstrations  $\mathcal{D}$  via ICL to sample code programs for a given language task instance. The  $\text{ICL}(\mathcal{D})$  operation creates a prompt composed of a program induction instruction and example-program pairs of the demonstration set. We are limited by context window size, so we condition on a randomly selected subset rather than all exemplars in  $\mathcal{D}$ . We initialize program induction with a hand-written set of domain-general exemplars. These exemplars, prompt instructions, and templates are shared in the Appendix B.

The terms in the denominator are a prior  $p(z_i|z_\ell)$  over code representations and a decoder  $p(x_i | z_i, z_\ell)$  from latent representation back to text. For code, the prior is established by a compiler that rewards syntactically valid programs:  $p(z_i | z_\ell) = \mathbb{1}(\text{compiles}(z))$ .

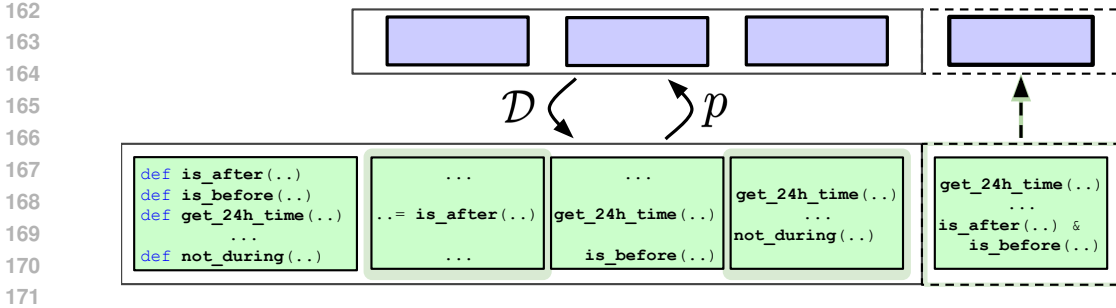


Figure 2: The underlying symbolic structure of observed text data (top) is represented by a shared function library and programs (bottom). We search for the set of demonstrations  $\mathcal{D}$  that optimizes the variational posterior. We sample from the induced posterior to synthesize new text (right).

**Algorithm 1** Codespace Autoencoder Pseudocode

```

177 procedure INFER( $x = \{x_i = (t_i, y_i)\}, \gamma$ )
178    $\mathcal{D} \leftarrow \{z_\ell^0, \emptyset\}$  ▷ initialize library and exemplars
179   for  $x_i = (t_i, y_i)$  do
180     for  $\tilde{z}_\ell, \tilde{z}_i \sim q$  do ▷ sample library additions, programs
181       if  $\neg \text{COMPILES}(\tilde{z}_i)$  then ▷ validate syntax
182         reject
183          $\tilde{z}_i(t_i) \leftarrow \text{EXEC}(\tilde{z}_i, t_i)$  ▷ execute program
184         if  $\text{textsim}(\tilde{z}_i(t_i), y_i) < \gamma$  then ▷ reject samples that fail reconstruction
185           reject
186            $\mathcal{D} \leftarrow \mathcal{D} \cup \{\tilde{z}_\ell, (x_i, \tilde{z}_i)\}$  ▷ add to library, save exemplar
187         break
188   return  $\mathcal{D}$ 

```

The decoder term  $p(x_i | z_i, z_\ell)$  measures how well the sampled code program reconstructs the observed data when executed with the given input. The term is decomposed into terms for the text input and output. The text input is given, so  $p(t | z) = 1$ .

$$p(x | z) = p(t | z) \times p(y | z, t) = p(y | z, t)$$

We approximate reconstruction accuracy  $p(y | z, t)$  with text similarity metric threshold  $\gamma$ :

$$p(y | z, t) = \begin{cases} 1, & \text{textsim}(z(t), y) > \gamma \\ \epsilon & \text{otherwise} \end{cases}$$

For reconstruction  $z(t)$ , we use a combination of a real interpreter and LLM interpretation.

To cover the scope of NLP tasks that cannot easily be performed programmatically, we allow sampling of programs with interleaved real code lines and imagined functions. We follow the **Chain-of-Code** (Li et al., 2024) method of using a Python interpreter as the real interpreter and a LLM as the imaginary interpreter. Specifically, for each line in a code program, we first attempt to execute it with a Python interpreter. If that fails, then we query a LLM to emulate the execution of that code line. The program state is then updated accordingly.

An overview of our method is sketched in Figure 2, and pseudocode presented in Algorithm 1.

**A Textspace Autoencoder for Language** To isolate the benefits of code as the latent representation, we consider another representation modality for the latent space: text. Here, we describe the autoencoder model framework in terms of a text-based latent representation: chain-of-thought (CoT) (Wei et al., 2022). Following the keyword prompting methodology of the TinyStories-Instruct (Eldan & Li, 2023) dataset, we describe the shared structure  $z_\ell$  as a library of unique keywords used by latent CoT strings of the dataset. Each  $(x_i, z_i)$  instance of  $\mathcal{D}$  is a paired language example and the corresponding CoT string. The  $\text{ICL}(\mathcal{D})$  operation creates a prompt in the same fashion, but using CoT-relevant domain-general exemplars. These exemplars and prompts are

shared in the Appendix B.2. The prior over CoT is established by a string checker that rewards strings with a certain CoT template:

$$p(z_i|z_\ell) = \mathbb{1}(z_i = \text{" * So the answer is * "})$$

Reconstruction accuracy is evaluated in the same way as for latent code representations, and the “execution” of the latent representation is string extraction.

## 5 POSTERIOR SAMPLING

Once we find a  $\mathcal{D}$  that optimizes our variational objective, we use it to sample more synthetic examples. Sampling new text from a given dataset is factorized in our autoencoder as follows:

$$\begin{aligned} p(x_s | x_{1:N}) &= \sum_z p(x_s | z, x_{1:N})p(z | x_{1:N}) \\ &= \mathbb{E}_{z \sim p(z|x_{1:N})} p(x_s | z, x_{1:N}) \\ &\approx \mathbb{E}_{z \sim q(z|x_{1:N}; \mathcal{D})} p(x_s | z, x_{1:N}) \\ &\approx \mathbb{E}_{z_\ell, z_s \sim q} p(x_s | z_s, z_\ell, x_{1:N}) \end{aligned}$$

The conditioning on  $x_{1:N}$  gives us the  $\mathcal{D}$  terms. This is used to sample our library  $z_\ell$  and to sample the program  $z_s$ . Since we do not have access to the observation  $x_s$  we sample the  $z_s$  using ICL that is unconditioned on the specific text datapoint.

Once sampled from the latent space, observed data  $x_s$  is decoded from  $z_s$ . A natural language input  $t_s$  is proposed for the sampled program, and the program  $z_s$  is executed to produce output  $y_s$ .

$$p(x_s | z_s, z_\ell, x_{1:N}) = p((t_s, y_s) | z_s, z_\ell, x_{1:N}) = p(t_s | z_s, z_\ell, \{t_i | (t_i, y_i) \in x_{1:N}\}) \times p(y_s | z_s, t_s)$$

## 6 EXPERIMENTAL SETUP

**Dataset: Super-NaturalInstructions (SNI)** (Wang et al., 2022) is a dataset of 1616 diverse NLP tasks, with expert-written instructions and over 5 million total data instances. The tasks cover 76 task types, some of which can be solved deterministically by a simple computer program (e.g. adding every third element of a list) and some of which are non-algorithmic natural language tasks. We use this categorization by algorithmic and non-algorithmic to study our method on a range of SNI tasks.

We divide the SNI training tasks into sets of 100 in-domain (ID) and 100 out-of-domain (OOD) tasks. Both sets of tasks are composed of 33% tasks with a “synthetic” (algorithmic) source. A latent space is induced for each ID task. Evaluation on the held-out examples from ID tasks comprise ID evaluation, and evaluation with the OOD tasks comprise OOD evaluation.

**Baselines:** We compare against two baseline methods. To measure the value of latent modeling, we compare our method against a direct inference (no intermediate reasoning) interpolation baseline. Interpolation uses LLM prompting to generate new data points based on an ICL prompt of the dataset text instances, and does not use a latent space. To measure the value of using code programs as the latent representation, we compare our method against using chain-of-thought (Wei et al., 2022) as the latent representation, as described in Section 4.

**Prompts:** All generations are produced by prompting the LLM with instructions and 4 in-context exemplars, or up to the maximum context length. Specific prompts are shared in the Appendix B.

We use 4 hand-written examples of input, programs, and output from the original Chain of Code paper (Li et al., 2024) as seed demonstrations to initialize the model fitting phase. Half of the examples are modified to use the internet search query `internet_lookup()` functionality that we add to LLM-emulation. These examples are shared in the Appendix B.1.

Latent Rep.	Recovery Rate (top-1 sampling)	
	Domain-general ICL	Induced $\mathcal{D}$ ICL
CoT	67.25	86.33
Code	53.75	70.67

Table 1: Autoencoding recovery rate increases when conditioned on the induced demonstrations  $\mathcal{D}$ . Measured across 1,200 instances across 100 different tasks; Llama3.1 8B model.

**Autoencoding Parameters:** We experiment with different demonstration set sizes  $N \in \{12, 24\}$ . We use 2 LLMs of different size: Mistral AI’s Mistral 8x22b Instruction-tuned model (Jiang et al., 2024) and Meta’s Llama3.1 8b Instruction-tuned model (Dubey et al., 2024). We use a temperature of 0.7 and top-p value of 0.9. For data generation, we sample  $M = \{40, 240\}$  latent representations from the posterior and decode them into text examples.

Executed latent representations are validated using ROUGE-L (Lin, 2004) and BLEU (Papineni et al., 2002) as the text similarity metrics, with corresponding thresholds  $\gamma_R = 0.4$  and  $\gamma_B = 0.3$ . During posterior sampling, code execution is validated by ICL-prompting a verifier LLM with examples from the demonstration set. Specific details are shared in Appendix A.

**LLM-Emulation Parameters:** Programs are executed with interleaved execution by the Python interpreter, `internet_lookup()` with the duckduckgo search engine API (ddg), and LLM-emulation with the generating model. The code to do this is slightly modified from that of the Chain of Code (Li et al., 2024) resources, and shared in our publicly-available repository <sup>1</sup>.

**Synthetic Data Generation Evaluation Training Parameters:** Evaluation of synthetically-generated data for downstream model training is measured by fine-tuning a Pythia 1.4B model (Biderman et al., 2023) on the synthesized text examples then evaluating the performance on the held-out test set. The model is trained with 12k steps, batch size 16, DeepSpeedFusedAdam optimizer, and learning rate of 1e-5.

## 7 RESULTS

In this section, we test the effectiveness of our method to identify the underlying symbolic space of a dataset. We measure this by evaluating first how well our symbolic representations are able to recover the original datasets. We then evaluate how well samples from the latent space cover the underlying task domain. Finally, we present results on using the latent space to generate synthetic training data.

### 7.1 FAITHFULNESS OF SYMBOLIC AUTOENCODING

We evaluate the extent to which the induced dataset structure and induced exemplars  $\mathcal{D}$  optimizes the variational posterior. Using a held-out  $x_j$  datapoint, we compare the autoencoder recovery rate when conditioning on our induced  $\mathcal{D}$  versus conditioning on a  $\mathcal{D}$  just of domain-general demonstrations, i.e. the hand-written domain-generic seed ICL demonstrations. Specifically we compute

$$(z_\ell, z_j) \sim q(z | \{x_j\}; \mathcal{D}) \quad \tilde{x}_j \sim p(x | z_j, z_\ell),$$

and compare  $\tilde{x}_j$  to  $x_j$ .

Table 1 shows that the variational method improves the autoencoding ability over the baseline method for code latent space. We also include evaluation using the CoT-based latent space, where inducing  $\mathcal{D}$  also leads to improved recovery.

### 7.2 EVALUATING THE IDENTIFIED SYMBOLIC SPACE

Our goal is to find a latent space from which more programs  $z_s$  can be effectively sampled and executed to produce new dataset examples. We compare against the latent CoT baseline, which uses

<sup>1</sup>released after anonymity period

Task type	Method	Correctness %		Domain Relevance %		Avg. cossim ( $\downarrow$ )
		Human	GPT-4o	Human	GPT-4o	
Algo.	Gold	-	60	-	100	0.81
Non-algo.	Gold	-	61.7	-	90	0.53
Llama3.1 8B						
Algo.	Sampling	61.7	18.3	65.0	48.3	<b>0.59</b>
	Latent CoT	50	20.0	93.3	<b>90</b>	0.89
	Latent Code	78.3	<b>50</b>	96.7	88.3	0.85
Non-algo.	Sampling	65.0	38.3	43.3	56.7	<b>0.52</b>
	Latent CoT	60	<b>43.3</b>	96.7	<b>85.0</b>	0.78
	Latent Code	81.7	36.7	68.3	45.0	0.76
Mixtral 8x22b						
Algo.	Sampling	-	20.0	-	<b>100</b>	0.94
	Latent CoT	-	38.3	-	98.3	0.94
	Latent Code	-	<b>60.0</b>	-	96.7	<b>0.88</b>
Non-algo.	Sampling	-	60.0	-	<b>100</b>	0.86
	Latent CoT	-	<b>63.3</b>	-	86.7	0.89
	Latent Code	-	45.0	-	91.7	<b>0.85</b>

Table 2: Human, GPT-4o-mini, and diversity evaluation of algorithmic and non-algorithmic text samples from different methods. For each method, we synthesize 240 examples. We examine 3 random algorithmic and 3 random non-algorithmic SNI tasks. Correctness and domain relevance is examined across 20 random samples per task. Diversity is measured with average embedding cosine similarity to centroid per task for all 240 synthesized instances.

a non-symbolic latent space composed of  $z_i$  as textual descriptions, and the interpolation baseline, which does not use a latent space.

We aim for examples that are correct, in-domain, and diverse. We evaluate this by sampling points as described in Section 5 for different algorithmic and non-algorithmic tasks. For correctness and domain relevance, we use GPT-4o-mini evaluation<sup>2</sup> for 20 examples per method per task, for 6 total tasks: 3 algorithmic and 3 non-algorithmic. Human evaluation numbers are provided for the Llama3.1 8B generations as a control against the GPT-4o-mini judgements. For diversity, we measure average cosine similarity between the Sentence-BERT embedding (Reimers & Gurevych, 2019) centroid for the synthesized dataset and each individual sentence embedding, a more tractable approximation for average pairwise cosine similarity used in prior works for diversity (Tevet & Berant, 2021). These results are summarized in Table 2.

Latent modeling methods produce more consistently domain-relevant and correct examples, though samples from the interpolation baseline become more domain-relevant with a larger model. Between using CoT and code as the latent representation, using code produces more correct data in algorithmic tasks and using CoT produces more correct data in non-algorithmic tasks. Diversity metrics, when observed with the domain relevance analysis, shows that the sampling baseline produces very diverse data to the point of being out-of-domain, and latent code produces slightly more diverse data than latent CoT.

### 7.3 SAMPLING SYNTHETIC TRAINING DATA

We compare the results of training a downstream model with data synthesized using our approach versus data synthesized by baseline methods. For these experiments, we train the downstream model on instances synthesized for a group of tasks then evaluate the resulting model on both in-domain and out-of-domain tasks. Table 3 summarizes the resulting performance. In our experiments, data from the sampling baseline generally outperforms data sampled from the latent modeling method. At times, data from the sampling baseline even outperforms Gold data provided by the original dataset.

<sup>2</sup>Prompt details in Appendix B.3

Synthesis Parameters				OOD ROUGE		ID ROUGE	
Method	Model	#Seeds	#Synth	Algo.	Nonalgo.	Algo.	Nonalgo.
Gold ; 40 examples			-	23.09	22.73	40.83	50.12
Sampling	L3.1 8B	24	40	22.89	31.07	31.77	40.66
	Mix 8x22b	24	40	23.89	31.24	<b>35.97</b>	<b>44.39</b>
Latent CoT	Mix 8x22b	24	40	21.92	30.56	34.05	39.14
Latent Code	L3.1 8B	24	40	<b>25.64</b>	28.33	33.22	38.62
	Mix 8x22b	24	40	21.73	<b>31.41</b>	34.22	39.94
Gold ; 240 examples			-	24.57	29.15	50.90	57.09
Sampling	L3.1 8B	12	240	<b>22.62</b>	<b>33.2</b>	35.08	44.04
	Mix 8x22b	12	240	20.93	30.43	<b>43.13</b>	<b>48.67</b>
Latent CoT	Mix 8x22b	12	240	20.41	29.23	30.75	39.43
Latent Code	L3.1 8B	12	240	21.72	30.33	32.65	38.22
	Mix 8x22b	12	240	22.12	30.21	35.64	36.91

Table 3: Downstream training results on SNI suggest that the interpolation baseline performs best, at times even yielding higher performance than using Gold data.

Model	Subset	% Programs Using Emulation	Avg. % LLM-emulated lines
L3.1 8B	Algorithmic	0	0
	Non-algorithmic	1.8	0.2
M8x22B	Algorithmic	2.0	0.2
	Non-algorithmic	17.8	2.6

Table 4: Induced latent code programs tend to rarely require LLM-emulation. Non-algorithmic tasks and larger models are more likely to leverage LLM-emulation.

## 8 QUALITATIVE ANALYSIS

**How often do we LLM-emulate?** Table 4 shows the rate of LLM-emulation in code programs in the induced demonstration set. The vast majority of programs are fully deterministically executable, but non-algorithmic tasks tend to use more lines of code that must be LLM-emulated. Larger models perform LLM-emulation more accurately (Li et al., 2024), and are more likely to propose lines that will be LLM-emulated.

An advantage of using code as the latent representation is the transparency of its execution. We can inspect the execution trace<sup>3</sup> to gain insight into what makes the latent code programs correct or incorrect. Example code programs from the induced demonstration sets, along with paired snippets from their execution trace, are shared below:

Listing 1: This program returns integers with certain conditions. This task can be done entirely programmatically.

```
def get_same_start_end(nums):
    result = []
    for num in nums:
        str_num = str(abs(num))
        if str_num[0] == str_num[-1] or (
            len(str_num) > 1 and str_num
            [0] > str_num[-1]):
            result.append(num)
    return result
```

```
[...]
line: result = []
explanation: Python execution.
delta state: {'answer': []}
line: for num in numbers:
explanation: Python execution.
delta state: {'num': -98}
line: str_num = str(abs(num))
explanation: Python execution.
delta state: {'str_num': '-98'}
line: if str_num[0] == str_num[-1] or (len(str_num)
    > 1 and str_num[0] > str_num[-1]):
explanation: Python execution.
delta state: {}
line: for num in nums:
explanation: Python execution.
delta state: {'num': 55}
[...]
final state: {'result': [44, 2, 98]}
```

<sup>3</sup>we follow the Chain-of-Code Li et al. (2024) template for LLM-emulated traces



Listing 2: To classify tweet toxicity, generated code imports and uses the NLTK sentiment analyzer.

```

import nltk
from nltk.sentiment import
    SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
def solve_task(task_input):
    comment = extract_comment(
        task_input)
    sia = SentimentIntensityAnalyzer()
    scores = sia.polarity_scores(
        comment)
    if scores['compound'] < -0.5:
        answer = "Yes"
    else:
        answer = "No"
    return answer

```

```

[...]
delta state: {'task_input': 'Categorize the comment
on the basis of toxicity. .... input: I hate
them'}
line: comment = extract_comment(task_input)
explanation: Let's think step by step. The
task_input variable has a full task description
, then a specified input... So the answer is "I
hate them"
delta state: {'comment': 'I hate them'}
line: sia = SentimentIntensityAnalyzer()
explanation: Python execution.
delta state: {'sia': nltk.sentiment.
SentimentIntensityAnalyzer object}
line: scores = sia.polarity_scores(comment)
explanation: Python execution.
delta state: {'scores': {'neg': 0.787, 'neu': 0.213,
'pos': 0.0, 'compound': -0.5719}}
[...]
final state: {'answer': 'Yes'}

```

Listing 3: To generate a question that every entity in an input list can be a valid answer to, generated code relies entirely on an imagined function: construct\_question

```

def solve_task(task_input):
    answers = task_input.split(", ")
    question = construct_question(
        answers, ret_type="str")
    return question

```

```

delta_state: {'task_input': 'Construct a question
that every answer in the list is correct and
reasonable for it. input: airplanes, snowflakes
, footballs, cards/valentines, paper dolls,
crane/bird'}
line: answers = task_input.split(", ")
explanation: Python execution
delta state: {'answer': ['Construct a question...
airplanes', 'snowflakes', 'footballs', 'cards/
valentines', 'paper dolls', 'crane/bird']}
line: question = construct_question(answers,
ret_tyep="str")
explanation: Let's think step-by-step. All of these
[...] So the answer is name something that kids
make out of paper
delta state: {'question': 'name something that kids
make out of paper'}

```

Does the shared structure of the latent space reflect underlying patterns of the task? Table 5 lists the most popular functions for some algorithmic and non-algorithmic SNI tasks. The most popular functions correspond with reasoning steps relevant to the task. For example, evaluating arithmetic under swapped operator strings mandates heavy use of string substitution (`re.sub`) and expression evaluation (`eval`). For less algorithmic tasks such as abductive reasoning in generating text to support health facts, functions related to inferring conclusions from facts such as `complete_story` and `generate_sentence` are popular. Hybrid tasks such as typo identification use low-level operations such as `word_tokenize` as well as high-level composed or imagined functions such as `get_typo`.

Task	Most frequently used functions
Get elements of list between given indices	<code>list_elements</code> , <code>get_elements</code>
Arithmetic under swapped operators	<code>eval</code> , <code>re.sub</code>
Identify typos	<code>word_tokenize</code> , <code>get_close_matches</code> , <code>get_typo</code>
Generate text to support health facts	<code>generate_claim</code> , <code>is_supported</code>
Write the middle sentence of a story	<code>complete_story</code> , <code>generate_sentence</code>

Table 5: Popular functions by task.

## 9 CONCLUSION

In this work, we present and apply a framework for extracting a latent symbolic representation from any text dataset in an unsupervised manner and without updating any model parameters. We use code programs as the latent representation, and present results comparing the resulting latent space to a non latent modeling approach and to using a text-based latent representation. Our findings

show that a codespace autoencoder for language tasks effectively extracts an underlying symbolic representation for language tasks. We show that code being inherently modular and executable offers advantages in correctness, particularly for algorithmic tasks, and lends well to interpreting the induced latent space. We believe that the methods and findings presented in this paper are applicable to future work in analyzing properties of datasets and generating data, both desired and undesired, and generating data for downstream tasks.

## ETHICS STATEMENT

One point of concern with this work is the ability to use the proposed methodology to generate foul or ill-harming text if given an input dataset exhibiting some of those patterns. We observed several concerning generations in tasks about classifying properties of toxic text, and caution any future practitioners to be aware of this potential danger.

## REPRODUCIBILITY STATEMENT

We take several steps towards reproducibility. First, we plan to release all code and prompts in a self-contained repository. Additionally, all experiments are run using open-sourced models and datasets that are publicly available on Huggingface. The code repository, publicly-available datasets, and open-source language models make almost all experiments of this paper reproducible. The only numbers that may be inconsistent across reproductions of the experiment are the human evaluation results and GPT-verification results of Table 2.

## REFERENCES

- duckduckgo-search. <https://pypi.org/project/duckduckgo-search/>. Accessed: 2024-09-24.
- Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O’Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. Pythia: a suite for analyzing large language models across training and scaling. In *Proceedings of the 40th International Conference on Machine Learning, ICML’23*. JMLR.org, 2023.
- Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. In *20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016*, pp. 10–21. Association for Computational Linguistics (ACL), 2016.
- Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL <https://openreview.net/forum?id=YfZ4ZPt8zd>.
- Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgina Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan

540 Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Ma-  
541 hadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy  
542 Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak,  
543 Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Al-  
544 wala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini,  
545 Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der  
546 Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo,  
547 Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Man-  
548 nat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova,  
549 Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal,  
550 Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur  
551 Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhar-  
552 gava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong,  
553 Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic,  
554 Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sum-  
555 baly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa,  
556 Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang,  
557 Sharath Rapparth, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende,  
558 Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney  
559 Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom,  
560 Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta,  
561 Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petro-  
562 vic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang,  
563 Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuwei Wang, Yaelle Goldschlag, Yashesh Gaur,  
564 Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Del-  
565 pierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha  
566 Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay  
567 Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda  
568 Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew  
569 Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita  
570 Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Barambe, Assaf Eisenman, Azadeh  
571 Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De  
572 Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Bran-  
573 don Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina  
574 Mejia, Changan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai,  
575 Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li,  
576 Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana  
577 Liskovich, Didem Foss, DingKang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil,  
578 Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Ar-  
579 caute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco  
580 Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella  
581 Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory  
582 Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang,  
583 Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Gold-  
584 man, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman,  
585 James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer  
586 Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe  
587 Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie  
588 Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun  
589 Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal  
590 Chawla, Kushal Lakhota, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva,  
591 Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian  
592 Khabba, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson,  
593 Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Ke-  
neally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel  
Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mo-  
hammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navy-  
ata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong,  
Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli,

- 594 Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux,  
595 Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao,  
596 Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li,  
597 Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott,  
598 Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Sa-  
599 tadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lind-  
600 say, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang  
601 Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen  
602 Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho,  
603 Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser,  
604 Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Tim-  
605 othy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan,  
606 Vinay Satish Kumar, Vishal Mangla, Vitor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu  
607 Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Con-  
608 stable, Xiaocheng Tang, Xiaofang Wang, Xiaojuan Wu, Xiaolan Wang, Xide Xia, Xilun Wu,  
609 Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi,  
610 Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef  
611 Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models, 2024.  
URL <https://arxiv.org/abs/2407.21783>.
- 612 Ronen Eldan and Yuanzhi Li. Tinstories: How small can language models be and still speak  
613 coherent english?, 2023. URL <https://arxiv.org/abs/2305.07759>.
- 614 Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt,  
615 Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping in-  
616 ductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIG-  
617 PLAN International Conference on Programming Language Design and Implementation, PLDI  
618 2021*, pp. 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN  
619 9781450383912. doi: 10.1145/3453483.3454080. URL [https://doi.org/10.1145/  
620 3453483.3454080](https://doi.org/10.1145/3453483.3454080).
- 621 Dan Friedman, Alexander Wettig, and Danqi Chen. Finding dataset shortcuts with grammar induc-  
622 tion. 2022.
- 623 Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris  
624 Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gi-  
625 anna Lengyel, Guillaume Bour, Guillaume Lample, Léo Renard Lavaud, Lucile Saulnier, Marie-  
626 Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le  
627 Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed.  
628 Mixtral of experts, 2024. URL <https://arxiv.org/abs/2401.04088>.
- 629 Diederik P Kingma. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- 630 Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen  
631 Chong, Gu-Yeon Wei, and Alexander M Rush. Guess & sketch: Language model guided tran-  
632 spilation. In *The Twelfth International Conference on Learning Representations*, 2024. URL  
633 <https://openreview.net/forum?id=qPFsIbF3V6>.
- 634 Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey  
635 Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-  
636 augmented code emulator. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller,  
637 Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st Interna-  
638 tional Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning  
639 Research*, pp. 28259–28277. PMLR, 21–27 Jul 2024. URL [https://proceedings.mlr.  
640 press/v235/li24ar.html](https://proceedings.mlr.press/v235/li24ar.html).
- 641 Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization  
642 Branches Out*, pp. 74–81, Barcelona, Spain, July 2004. Association for Computational Linguis-  
643 tics. URL <https://www.aclweb.org/anthology/W04-1013>.
- 644 Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language mod-  
645 els of code are few-shot commonsense learners. In Yoav Goldberg, Zornitsa Kozareva, and  
646

- 648 Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Lan-*  
649 *guage Processing*, pp. 1384–1403, Abu Dhabi, United Arab Emirates, December 2022. Asso-  
650 ciation for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.90. URL <https://aclanthology.org/2022.emnlp-main.90>.  
651
- 652 Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David  
653 Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Au-  
654 gustus Odena. Show your work: Scratchpads for intermediate computation with language models,  
655 2021. URL <https://arxiv.org/abs/2112.00114>.  
656
- 657 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic  
658 evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association*  
659 *for Computational Linguistics*, ACL ’02, pp. 311–318, USA, 2002. Association for Computa-  
660 tional Linguistics. doi: 10.3115/1073083.1073135. URL [https://doi.org/10.3115/](https://doi.org/10.3115/1073083.1073135)  
661 [1073083.1073135](https://doi.org/10.3115/1073083.1073135).  
662
- 663 Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Mea-  
664 suring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan  
665 Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP*  
666 *2023*, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics.  
667 doi: 10.18653/v1/2023.findings-emnlp.378. URL [https://aclanthology.org/2023.](https://aclanthology.org/2023.findings-emnlp.378)  
668 [findings-emnlp.378](https://aclanthology.org/2023.findings-emnlp.378).  
669
- 670 Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin  
671 Wang, Yoon Kim, Yejin Choi, Nouha Dziri, and Xiang Ren. Phenomenal yet puzzling: Testing  
672 inductive reasoning capabilities of language models with hypothesis refinement. In *The Twelfth*  
673 *International Conference on Learning Representations*, 2024. URL [https://openreview.](https://openreview.net/forum?id=bNt7oajl2a)  
674 [net/forum?id=bNt7oajl2a](https://openreview.net/forum?id=bNt7oajl2a).  
675
- 676 Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-  
677 networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language*  
678 *Processing*. Association for Computational Linguistics, 11 2019. URL [https://arxiv.](https://arxiv.org/abs/1908.10084)  
679 [org/abs/1908.10084](https://arxiv.org/abs/1908.10084).  
680
- 681 Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro,  
682 Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can  
683 teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing*  
684 *Systems*, 2023. URL <https://openreview.net/forum?id=Yacmpz84TH>.  
685
- 686 Guy Tevet and Jonathan Berant. Evaluating the evaluation of diversity in natural language genera-  
687 tion. In Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty (eds.), *Proceedings of the 16th Confer-*  
688 *ence of the European Chapter of the Association for Computational Linguistics: Main Volume*,  
689 pp. 326–346, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/  
690 [2021.eacl-main.25](https://aclanthology.org/2021.eacl-main.25). URL <https://aclanthology.org/2021.eacl-main.25>.  
691
- 692 Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei,  
693 Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Es-  
694 haan Pathak, Giannis Karamanolakis, Haizhi Lai, Ishan Purohit, Ishani Mondal, Jacob An-  
695 derson, Kirby Kuznia, Krma Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi,  
696 Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravse-  
697 haj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan  
698 Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-NaturalInstructions: Generaliza-  
699 tion via declarative instructions on 1600+ NLP tasks. In Yoav Goldberg, Zornitsa Kozareva,  
700 and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natu-*  
701 *ral Language Processing*, pp. 5085–5109, Abu Dhabi, United Arab Emirates, December 2022.  
Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.340. URL  
<https://aclanthology.org/2022.emnlp-main.340>.
- Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes  
for solving programmatic tasks, 2024. URL <https://arxiv.org/abs/2401.12869>.

702 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V  
 703 Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models.  
 704 In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in*  
 705 *Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc.,  
 706 2022. URL [https://proceedings.neurips.cc/paper\\_files/paper/2022/](https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf)  
 707 [file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf).

708 Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. Parsel : Algo-  
 709 rithmic reasoning with language models by composing decompositions, 2022a. URL <https://arxiv.org/abs/2212.10561>.  
 710

711 Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STar: Bootstrapping reasoning with  
 712 reasoning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.),  
 713 *Advances in Neural Information Processing Systems*, 2022b. URL [https://openreview.](https://openreview.net/forum?id=_3ELRdg2sgI)  
 714 [net/forum?id=\\_3ELRdg2sgI](https://openreview.net/forum?id=_3ELRdg2sgI).  
 715

716 Wenting Zhao, Justin Chiu, Claire Cardie, and Alexander Rush. Abductive commonsense reason-  
 717 ing exploiting mutually exclusive explanations. In Anna Rogers, Jordan Boyd-Graber, and  
 718 Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Com-*  
 719 *putational Linguistics (Volume 1: Long Papers)*, pp. 14883–14896, Toronto, Canada, July  
 720 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.831. URL  
 721 <https://aclanthology.org/2023.acl-long.831>.

722 Wenting Zhao, Justin Chiu, Claire Cardie, and Alexander Rush. Hop, union, generate: Explainable  
 723 multi-hop reasoning without rationale supervision. In Houda Bouamor, Juan Pino, and Kalika  
 724 Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language*  
 725 *Processing*, pp. 16119–16130, Singapore, December 2023b. Association for Computational Lin-  
 726 guistics. doi: 10.18653/v1/2023.emnlp-main.1001. URL [https://aclanthology.org/](https://aclanthology.org/2023.emnlp-main.1001)  
 727 [2023.emnlp-main.1001](https://aclanthology.org/2023.emnlp-main.1001).  
 728

## 729 A VERIFYING SYNTHESIZED EXAMPLES

730  
 731 A challenge of synthesizing correct examples is the lack of a ground truth output against which to  
 732 verify the execution of the sampled latent code program. For this, we parameterize a verifier LLM  
 733 with the learned demonstration set  $\mathcal{D}$  by collecting seed demonstrations  $(z, x, y)$  according to their  
 734 reconstruction success  $p(x, y | z)$ . Successful reconstructions are saved as positive examples  $v^+ =$   
 735  $\{(z, x, y : z(x) \equiv y)\}$  and failed reconstructions as negative  $v^\times = \{(x, y_x, z_x : z_x(x) \not\equiv y_x)\}$ .  
 736 The verifier LLM is parameterized using ICL with these positive and hard negative examples to  
 737 predict the correctness of an input demonstration:

$$738 p(x_s, y_s, z_s | \mathcal{D}) := p(x_s, z_s(x_s), z_s | v^+, v^\times)$$

## 740 B PROMPTS

### 741 Listing 4: Program induction instruction

742  
 743  
 744 Write the Python code to perform the given task. The ‘task\_output’  
 745 variable at the end should contain the response to the  
 746 instruction.

### 747 Listing 5: Input generation instruction

748  
 749 Write the natural language instruction that describes what is done  
 750 by the given Python code. Provide the instruction and all  
 751 relevant inputs.

### 752 Listing 6: Instruction for posterior sampling of latent code programs

753  
 754 Write a snippet of Python code to complete some task. The code  
 755 should use the following functions: {keywords}.

## B.1 HAND-WRITTEN EXEMPLAR DEMONSTRATIONS

Listing 7: Demo examples for latent code variational inference.

```

756
757
758
759
760
761 Which sentence has the correct adjective order:
762 Options:
763 (A) rubber terrible ship
764 (B) terrible rubber ship
765 --> (B)
766 # CODE START
767 import numpy as np
768 def solve_task(task_input):
769     options = extract_options(task_input)
770     priority = {"opinion": 1, "size": 2, "age": 3, "shape": 4, "
771               color": 5, "origin": 6, "material": 7, "purpose": 8}
772     valid_types = list(priority.keys())
773     scores = []
774     for option, sentence in options.items():
775         adjectives = sentence.split(" ")[::-1]
776         order = [priority[get_adjective_type(adjective, valid_types,
777         ret_type=str)] for adjective in adjectives]
778         scores.append([order[i+1] > order[i] for i in range(len(order)
779         - 1)].count(True))
780     answer = list(options.keys())[np.argmax(scores)]
781     return answer
782 # CODE END
783
784 How many countries have I been to? I've been to Bilbao, Death
785 Valley, Paris, Honolulu, Skye.
786 --> 4
787 # CODE START
788 def solve_task(task_input):
789     places = extract_places(task_input)
790     countries = set()
791     for place in places:
792         search_result = lookup_on_internet(f"what country is {place}
793         in?")
794         country = extract_country(search_result)
795         countries.add(country)
796     answer = len(countries)
797     return answer
798 # CODE END
799
800 Today is Christmas Eve of 1937. What is the date 10 days ago in MM
801 /DD/YYYY?
802 --> 12/14/1937
803 # CODE START
804 import datetime
805 def solve_task(task_input):
806     parsed_task_input = extract_arguments(task_input)
807     today = datetime.date(year=parsed_task_input['year'], month=
808     parsed_task_input['month'], day=parsed_task_input['day'])
809     date = today - datetime.timedelta(days=parsed_task_input['
810     days_ago'])
811     answer = date.strftime("%m/%d/%Y")
812     return answer
813 # CODE END

```

```

810 Recommend a movie similar to Star Wars Episode IV - A New Hope,
811 Indiana Jones and the Last Crusade, Star Wars Episode V - The
812 Empire Strikes Back, The Big Lebowski:
813 --> Blade Runner (1982)
814 # CODE START
815 def solve_task(task_input):
816     ref_movies = extract_reference_movies(task_input)
817     ref_movie_infos = []
818     for movie in ref_movies:
819         search_result = lookup_on_internet(f"{movie} release year and
820 genre")
821         release_year = extract_year(search_result)
822         genre = extract_genre(search_result)
823         ref_movie_infos.append((genre, release_year))
824     answer = get_most_similar_movie(ref_movies, ref_movie_infos,
825 ret_type="str")
826     return answer
827 # CODE END

```

## 828 B.2 CHAIN-OF-THOUGHT PROMPTS

### 830 Listing 8: CoT induction instruction

832 Write the thought process to perform the given task.

### 834 Listing 9: Input generation instruction for latent CoT

835 Write the natural language instruction that describes what is  
836 shown by the given thought process. Provide the instruction  
837 and all relevant inputs.

### 840 Listing 10: Instruction for posterior sampling of CoT

841 Write out the thought process that completes some task. The  
842 thought process should include the following terms: {keywords  
843 }.

### 845 Listing 11: Demo examples for variational inference of latent CoT.

846 Which sentence has the correct adjective order:  
847 Options:  
848 (A) rubber terrible ship  
849 (B) terrible rubber ship  
850 --> (B)  
851 Let's think step by step.  
852 The priority of adjective categories is, in order: opinion, size,  
853 age, shape, color, origin, material, purpose.  
854 rubber: material  
855 terrible: opinion  
856 The opinion adjective should come before the material adjective. '  
857 terrible rubber ship' follows the correct adjective order. So  
858 the answer is (B).

859 How many countries have I been to? I've been to Bilbao, Death  
860 Valley, Paris, Honolulu, Skye.  
861 --> 4  
862 Let's think step by step.  
863 We'll group by countries and count:  
1. Spain: Bilbao



864 2. USA: Death Valley, Honolulu  
 865 3. France: Paris  
 866 4. UK: Skye  
 867 There are 4 countries in total. So the answer is 4.  
 868  
 869 Today is Christmas Eve of 1937. What is the date 10 days ago in MM  
 870 /DD/YYYY?  
 871 --> 12/14/1937  
 872 Let's think step by step.  
 873 Christmas Eve of 1937 is 12/24/1937.  
 874 To get the date 10 days ago, subtract 10 from the date section. So  
 875 the answer is 12/14/1937.  
 876  
 877 Recommend a movie similar to Star Wars Episode IV - A New Hope,  
 878 Indiana Jones and the Last Crusade, Star Wars Episode V - The  
 879 Empire Strikes Back, The Big Lebowski:  
 880 --> Blade Runner (1982)  
 881 Let's think step by step.  
 882 The genre and release year of each of the reference movies are:  
 883 Star Wars Episode IV - A New Hope: Sci-fi/Action (1977)  
 884 Indiana Jones and the Last Crusade: Adventure/Action (1989)  
 885 Star Wars Episode V - The Empire Strikes Back: Family/Sci-fi  
 886 (1980)  
 887 The Big Lebowski: Comedy/Crime (1998)  
 888 A similar movie should have a release date in 1970-2000 and be Sci  
 889 -fi/Action/Crime. So the answer is Blade Runner (1982).  
 890

### 891 B.3 MEASURING DATA QUALITY WITH GPT-4O-MINI

#### 892 Listing 12: Instruction prompt to verify data quality

895 You are tasked with evaluating the following instances for correctness  
 896 and domain relevance.  
 897  
 898 Please:  
 899 1. In [CORRECT] and [/CORRECT] tags, state if the output seems correct  
 900 based on the input and domain task description. Write Yes or No.  
 901 2. In [RELEVANT] and [/RELEVANT] tags, assess if the input and output are  
 902 relevant to the specified domain (e.g., the context of the  
 903 experiment). Write Yes or No.  
 904 Provide your reasoning for each answer, if necessary.  
 905  
 906 Domain task description: Provide a movie recommendation.  
 907 Input: Recommend a movie similar to Star Wars Episode IV - A New Hope,  
 908 Indiana Jones and the Last Crusade, Star Wars Episode V - The Empire  
 909 Strikes Back, The Big Lebowski:  
 910 Output: Blade Runner (1982)  
 911 1. All of these movies are action movies released around 1990 to 2010. So  
 912 Blade Runner (1982) is an appropriate recommendation. [CORRECT]Yes[/  
 913 CORRECT]  
 914 2. The task is to provide a movie recommendat. This is domain-relevant. [  
 915 RELEVANT]Yes[/RELEVANT]  
 916  
 917 Domain task description: Identify whether the given comment is severely  
 918 toxic.  
 919 Input: Sammy wanted to go to where the people were. Where might he go?  
 920 Options: (a) race track (b) populated areas (c) desert (d) apartment  
 921 (e) roadblock  
 922 Output: (b)

918  
 919 1. People are at populous places, so if Sammy wants to go where the  
 920 people are, he should go there. Option (b) is correct. [CORRECT]Yes[/  
 921 CORRECT]  
 922 2. The task is to identify whether a given comment is severely toxic.  
 923 Determining where Sammy wants to go has nothing to do with  
 924 identifying comment toxicity. [RELEVANT]No[/RELEVANT]  
 925 Domain task description: {task\_description}  
 926 Input: {instance\_input}  
 927 Output: {instance\_output}

## 928 929 C SYNTHESIZED EXAMPLES

930  
 931 Examples generated by the sampling baseline tend to be easier, out-of-domain, or incorrect. This  
 932 is likely because sampling simply relies on surface-level generation and prefers generating strings  
 933 with low perplexity, i.e. it is comfortable with.  
 934

### 935 Listing 13: Mixtral 8x22b Sampling (incorrect)

936 In this task you will be given a list of integers. You should only  
 937 return an integer if the first digit is the same as the last  
 938 digit in the number. If an integer has a single digit, then it  
 939 should be returned. If there are no integers that start and  
 940 end with the same digit then an empty list ("[]") should be  
 941 returned.  
 942 input: [989, 597, 930, 637, 648, 647, -972, 943, 269, -486, 530,  
 943 974, 993, -631, 431, 979, -701, 560, -928, 935]  
 944 output: [989, 993]

### 945 946 Listing 14: Llama3 8.1b Sampling (very easy, OOD)

947  
 948 In this task you will be given a list of integers. You should only  
 949 return an integer if the first digit is the same as the last  
 950 digit in the number. If an integer has a single digit, then it  
 951 should be returned. If there are no integers that start and  
 952 end with the same digit then an empty list ("[]") should be  
 953 returned.  
 954 input: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
 955 output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

956  
 957 Examples generated by the latent CoT baseline are good at generating in-domain data, but are often  
 958 wrong in algorithmic settings.

### 959 Listing 15: Llama3 8.1b Latent CoT Baseline (non-algorithmic, correct)

960  
 961 In this task, you are given a description of a person (PersonX)  
 962 and a description of an event. You have to determine whether  
 963 the description of the person necessarily implies the  
 964 description of the event, or whether the description of the  
 965 person may imply the description of the event, or whether the  
 966 description of the person does not necessarily imply the  
 967 description of the event. Classify your answers into "Yes",  
 968 "No", and "Maybe". The description of the person and the  
 969 event may also contain "\_\_\_\_", a placeholder that can be an  
 970 object, a person, and/or an action.  
 971 input: PersonX: PersonX bakes a cake<sep>Event: PersonX  
 accidentally drops the cake  
 COT: Lets think step by step.

972 1. The description of PersonX is that they bake a cake. The  
 973 description of the event is that they accidentally drop the  
 974 cake. The two concepts do not match because the description of  
 975 the event does not imply that PersonX has a certain quality,  
 976 which is baking a cake.

977 3. The description of PersonX is that they bake a cake. The  
 978 description of the event is that they accidentally drop the  
 979 cake. The two concepts do not match because the description of  
 980 the event does not imply that PersonX has a certain quality,  
 981 which is baking a cake.

982 Therefore, the description of PersonX does not necessarily imply  
 983 the description of the event.

984 However, the description of PersonX may imply the description of  
 985 the event, because the description of the event is a direct  
 986 result of the description of PersonX. The description of  
 987 PersonX is that they bake a cake. The description of the event  
 988 is that they accidentally drop the cake. The two concepts may  
 989 match because the description of the event is a direct result  
 990 of the description of PersonX. Therefore, the description of  
 991 PersonX may imply the description of the event.

992 However, the description of PersonX does not necessarily imply the  
 993 description of the event. The description of PersonX is that  
 994 they bake a cake. The description of the event is that they  
 995 accidentally drop the cake. The two concepts do not match  
 996 because the description of the event does not imply that  
 997 PersonX has a certain quality, which is baking a cake.

998 Therefore, the description of PersonX does not necessarily imply  
 999 the description of the event.

1000 However, the description of PersonX may imply the description of  
 1001 the event, because the description of the event is a direct  
 1002 result of the description of PersonX. The description of  
 1003 PersonX is that they bake a cake. The description of the event  
 1004 is that they accidentally drop the cake. The two concepts may  
 1005 match because the description of the event is a direct result  
 1006 of the description of PersonX. Therefore, the description of  
 1007 PersonX may imply the description of the event.

1008 However, the description of PersonX does not necessarily imply the  
 1009 description of the event.

1010 So the answer is Maybe.

#### Listing 16: Llama3 8.1b Latent CoT Baseline (algorithmic, wrong)

1011 In this task, you need to remove all words of a given list in the  
 1012 sentence. The list of words to be removed is given, for  
 1013 example, the list ['bear', 'scissors', 'beach']. Ignore the  
 1014 words that are not present in the sentence.

1015 input: Sentence: 'a group of people are playing with food on the  
 1016 beach near a dark place hanging by a seat on the ocean with  
 1017 sand near a computer'. Remove all words in the given list: ['  
 1018 bear', 'scissors', 'beach'].

1019 input: a group of people are playing with food near a dark place  
 1020 hanging by a seat on the ocean with sand near a computer  
 1021 output: Let's think step by step.

1022 The 3 words to remove are 'bear', 'scissors', 'beach'.  
 1023 However, 'beach' is mentioned several times in the sentence, so we  
 1024 have to remove the first 'beach'.  
 1025 The sentence is: 'a group of people are playing with food on the  
 beach near a dark place hanging by a seat on the ocean with  
 sand near a computer'.

1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079

So the result is 'a group of people are playing with food near a dark place hanging by a seat on the ocean with sand near a computer'.  
So the answer is: a group of people are playing with food near a dark place hanging by a seat on the ocean with sand near a computer.