An Iterative Bayesian Approach to Premise Generation with Entailment Trees

Anton Chen [1]

Abstract

In the context of question answering (QA), entailment trees represent question-answer relationships by modelling intermediate premises with tree structure. We propose an iterative Bayesian approach to modelling the distribution of entailment trees with the specific inference task of sampling for new premises. We construct a probabilistic model on entailment trees to analyze the ENTAILMENTBANK dataset, a set of 1 840 tree-structured explanations to science exam questions. More specifically, our model iteratively retrieves and generates premises to construct entailment trees bottom-up from the leaf nodes. We explore approximate techniques of Markov Chain Monte Carlo (MCMC) and stochastic variational inference (SVI) in PYRO, a probabilistic programming language (PPL) for PYTHON, along with implementations for relevant data pre-processing utilities.

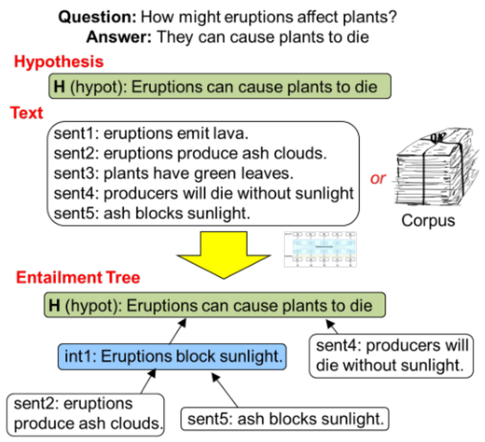
Question: How might eruptions affect plants?
Answer: They can cause plants to die
Hypothesis
H (hypot): Eruptions can cause plants to die
Text
sent1: eruptions emit lava.
sent2: eruptions produce ash clouds.
sent3: plants have green leaves.
sent4: producers will die without sunlight
sent5: ash blocks sunlight.
or
Corpus
Entailment Tree
H (hypot): Eruptions can cause plants to die
int1: Eruptions block sunlight.
sent4: producers will die without sunlight.
sent2: eruptions produce ash clouds.
sent5: ash blocks sunlight.

Figure 1. Example of an entailment tree generated from an answer and rationale. `sent2` and `sent5` are used to generate `int1`, and then `int1` and `sent4` generate `hypot`. This example was taken from Dalvi et al. and is in the ENTAILMENTBANK dataset. (2021).

1. Introduction

In the context of Natural Language Processing, automated question answering (QA) stands as an important task with widespread applications. QA systems aim to not only automate the process of generating accurate answers, but also exhibit interpretability via intermediate reasoning, i.e. how we got to the answer. To elaborate, a question-answer relationship is not simply binary — answers often arise from intermediate context and premises rather than directly from the original question prompt. When humans solve difficult problems, we don't immediately spit out an answer. We write things out step by step, and oftentimes, each step depends on multiple previous steps.

Entailment trees, initially introduced by Dalvi et al., represent the line of reasoning associated with answering a question via tree-structured premises (2021). Entailment trees express the ability to model question-answers pairs in more detail than other approaches, such as rationales (DeYoung et al., 2019) or multi-hop reasoning chaining (Jhamtani & Clark, 2020).

The key problem that this paper is interested in is premise generation: given a set of premises, generate new premises (i.e. parent nodes of a subset of candidate premises). *The learning goal of this project is to perform Bayesian inference over the non-standard data type of tree-structured entailment data for the purpose of tree generation.*

2. Related Work

The original paper on entailment trees by Dalvi et al. not only introduces the dataset as a benchmark for language models, but also propose the T5-based ENTAILMENTWRITER architectures as baselines for tree generation (2021).

In response, a few approaches have risen for entailment tree generation. Ribero et al. propose the ITERATIVE

[1]Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada. Correspondence to: Anton Chen <contact@antonchen.ca>.

STAT 447C *Bayesian Statistics*, 2024W2. Source code is available at https://github.com/chenanton/entailment-trees

RETRIEVAL-GENERATION REASONER (IRGR) architecture, which given a hypothesis and a set of textual premises, iteratively generates an entailment tree by first retrieving premises, and then generating a premise given the retrieved context — one premise at a time (Ribeiro et al., 2022). Hong et al. propose METGEN, an architecture which leverages multiple entailment modules (e.g. substitution, conjugation) and a reasoning controller, also iteratively generating the entailment tree (Hong et al., 2022). A common theme we see here is iterative tree generation: premises are generated one at a time.

This paper follows in the spirit of iterative tree generation, but aims to differ from related work. Firstly, our model aims to replace transformer-based sequence-to-sequence models with a simple linear model in embedding space to observe how performance is affected.

Secondly, our model differs from others by expanding upon the premise retrieval process. To elaborate, the IRGR-RETRIEVER module of IRGR statically defines premise retrieval probability with a method known as *dense passage retrieval* (Karpukhin et al., 2020), assigning retrieval probability proportional to the semantic similarity to the hypothesis (Ribeiro et al., 2022). As self-admittedly "leaf sentences are not always directly related to hypothesis" (Ribeiro et al., 2022), our model attempts to relax this assumption by parameterizing the premise retrieval distribution. As a result, our retrieval probability distribution does not explicitly follow dense passage retrieval and utilizes a prior instead.

Finally, our inference task is slightly different: tree generation from premises without a target hypothesis. The motivation is this: in practice, given some premises, the answer ultimately entailed is unknown: oftentimes we don't know the answer before we know the facts.

3. Methodology

The focus of this section is to construct a probability model for entailment trees. We'll first begin by formulating the problem of entailment tree generation as a concrete inference problem.

To model the distribution of entailment trees, we take some inspiration from related work: we assume that entailment tree construction is an *iterative* process (Ribeiro et al., 2022; Hong et al., 2022). To elaborate, new premises are generated one-by-one, conditioned on a subset of available premises. All premises are initially available, and become unavailable once they are used to entail another new premise. Newly generated premises are added to the set of available premises. This iterative update pattern constructs the tree bottom-up. We'll spend this section to formalize this intuition.

3.1. Problem Formulation

The observed inputs of the model are the entailment trees in our dataset y_i for $i = 1, \cdots, n$ where n is the size of our dataset. Each y_i has an initial set of premises $S_i^{(0)} = \{s_{i1}^{(0)}, \cdots s_{im_i^{(0)}}^{(0)}\}$, which are the leaf nodes of y_i. In words the ith example has $m_i^{(0)}$ initial premises, where each premise is a sentence.

Rather than direct word representation, we employ sentences embeddings by encoding premises into dense continuous vectors of fixed lower dimension. Embeddings algorithms such as DOC2VEC and SENTENCE-BERT have found major success in other domains of language processing (Le & Mikolov, 2014; Reimers & Gurevych, 2019). Mathematically, let $\phi\colon \mathcal{S} \to \mathbb{R}^d$ be a sentence encoder (where $\mathcal{S}$ is the sentence space and d is the embedding dimension). We say $c \triangleq \phi(s)$ is the encoding of s.

From this we arrive at our encoded dataset $C^{(0)} \triangleq (C_1^{(0)}, \cdots, C_n^{(0)})$ where $C_i^{(0)} \triangleq [c_{i0}^{(0)}, \cdots, c_{im_i^{(0)}}^{(0)}]^\top \in \mathbb{R}^{m_i^{(0)} \times d}$ and $c_{ij}^{(0)} \triangleq \phi(s_{ij}^{(0)}) \in \mathbb{R}^d$. We use this notation below to refer to the embeddings of sentences where necessary.

The main interest of our model is an inference task: given a set of initial premises $C_{\text{test}}^{(0)} \in \mathbb{R}^{m_{\text{test}}^{(0)} \times d}$, sample an entailment tree y_{test} from the posterior distribution $\mathbb{P}[y_{\text{test}} \mid y, S_{\text{test}}^{(0)}]$. The tree y_{test} represents generated premises w.r.t. initial premises $S_{\text{test}}^{(0)}$. However to limit the scope and computational burden of this project, our model will only be for the posterior distribution a single generated premise rather than the entire tree, i.e. the posterior $\mathbb{P}[c_{\text{test}}^{*(0)}, \tilde{c}_{\text{test}}^{(0)} \mid y, S_{\text{test}}^{(0)}]$. Note that $\tilde{c}_{\text{test}}^{(0)}$ is included so the retrieved premises used to generate $c_{\text{test}}^{*(0)}$ are known.

3.2. Probability Model

Let t be the tth iteration in this process. The total number of iterations is dependent on the number of premises retrieved at each step. Let $S_i^{(t)}$ be the set of available premises to be used for generation at the tth iteration for sample i. $C_i^{(t)}$ represents the encoded sentences corresponding to $S_i^{(t)}$.

3.2.1. PREMISE RETRIEVAL

Premise retrieval is the task of selecting a subset of available premises $\tilde{S}_i^{(t)} \subset S_i^{(t)}$ used to generate a new premise. Let $k_i^{(t)} \triangleq |\tilde{S}_i^{(t)}|$ be the number of premises retrieved at iteration t. For example, in Figure 1, $\tilde{s}_i^{(0)} = \{\texttt{sent2}, \texttt{sent5}\}$ and $\tilde{s}_i^{(1)} = \{\texttt{int1}, \texttt{sent4}\}$ with $k_i^{(0)} = k_i^{(1)} = 2$. Our model's first step is to sample for the number of premises used to generate a new premise.

The model utilizes the prior and likelihood

$$p \sim \text{Uni}(0,1), \tag{1}$$

$$k_i^{(t)} - 1 \sim \text{Binomial}(m_i^{(t)} - 1, p), \tag{2}$$

which is well-motivated as $k_i^{(t)}$ has support $\{1, 2, \cdots, m_i^{(t)}\}$. Next we retrieve $k_i^{(t)}$ premises. Ribeiro et al. make use of *dense passage retrieval*, a method which explicitly defines retrieval probabilities of each premise based on similarity to the hypothesis and available premises. We relax this assumption by adapting this method into a linear model. For $W_r \in \mathbb{R}^{d\times d}$ with prior

$$(W_r)_{xy} \sim \mathcal{N}(0,1) \qquad \text{(For } x, y = 1, \cdots, d\text{)}$$

the conditional retrieval probabilities

$$p_{ij}^{(t)} \triangleq \mathbb{P}\left[\tilde{c}_{ij}^{(t)} \mid C_i^{(t)}\right] = \frac{\exp\left(\psi\left(\tilde{c}_{ij}^{(t)}, C_i^{(t)}\right)\right)}{\sum_{c' \in C_i^{(t)}} \exp\left(\psi\left(c', C_i^{(t)}\right)\right)} \tag{3}$$

where

$$\psi(c, C) = c^\top W_r \left(\frac{1}{m_i^{(t)}} C^\top \mathbf{1}_{m_i^{(t)}}\right)$$

represents a linear combination of the candidate premise and the mean available premise. In Ribeiro et al.'s paper, ψ corresponds to the inner product between a candidate premise and the hypothesis, whereas our proposed model learns ψ. Again the prior on W_r is well-motivated as its support is $\mathbb{R}^{d\times d}$.

Now the premises to be retrieved can be sampled with a categorical distribution. The $k_i^{(t)}$ retrieved premises must be distinct, so the premises are sequentially sampled to ensure each are distinct from each other. Mathematically,

$$\begin{aligned}
j_{i1}^{(t)} &\sim \text{Cat}\left(\theta_{i1}^{(t)}\right) \text{ where } \theta_{i1}^{(t)} = \left(p_{i1}^{(t)}, \cdots, p_{im_i}^{(t)}\right) \\
j_{i2}^{(t)} \mid j_{i1}^{(t)} &\sim \text{Cat}\left(\theta_{i2}^{(t)}\right) \text{ where} \\
&\quad \theta_{i2}^{(t)} \propto \left(p_{i1}^{(t)}, \cdots, p_{i(j_{i1}^{(t)}-1)}^{(t)}, 0, \cdots, p_{im_i}^{(t)}\right), \\
&\quad \vdots
\end{aligned}$$

until we have sampled $j_{i1}^{(t)}, \cdots, j_{ik_i^{(t)}}^{(t)}$. With this model in place, sampling yields retrieved premises $\tilde{C}_i^{(t)} \triangleq (C_i^{(t)})_{j_{i1}^{(t)}}, \cdots, (C_i^{(t)})_{j_{ik_i^{(t)}}^{(t)}}$. Apologies for the notation here. In words, we sample $k_i^{(t)}$ distinct columns of $C_i^{(t)}$. For shorthand, denote $\tilde{C}_i^{(t)} \sim \text{Retrieve}(W_r, k_i^{(t)}, C_i^{(t)})$.

3.2.2. Premise Generation

Now that we have determined which premises $\tilde{C}_i^{(t)}$ will generate the new premise, let's proceed with generating the new premise. For $W_g \in \mathbb{R}^{d\times d}$, priors

$$\begin{aligned}
(W_g)_{xy} &\sim \mathcal{N}(0,1) \qquad \text{(For } x, y = 1, \cdots, d\text{)} \\
\sigma^2 &\sim \text{Exp}(1),
\end{aligned}$$

and mean retrieved premise

$$\bar{c}_i^{(t)} = \frac{1}{k_i^{(t)}} \tilde{C}_i^{(t)\top} \mathbf{1}_{k_i^{(t)}},$$

we represent the conditional premise distribution with a linear model:

$$c_i^{*(t)} \mid \bar{c}_i^{(t)} \sim \mathcal{N}\left(W_g \bar{c}_i^{(t)}, I_d \sigma^2\right)$$

where $\mathbf{0}_d$ and I_d denote the d-dimensional zero vector and $d \times d$ identity matrix respectively. It is worth noting that our model assumes a unimodal conditional distribution, i.e. given a set of premises, there's a singular standout entailment. The model also assumes homoscedasticity and independent variance between features in embedding space. This is a key limitation of this model, and further exploration can be done to explore a different set of assumptions (e.g. heteroscedasticity).

3.2.3. Update Available Premises

Now that a premise has been sampled, the set of available premises must be updated for the next iteration. Formally

$$S_i^{(t+1)} \triangleq \left\{s \in S_i^{(t)} : s \notin \tilde{S}_i^{(t)}\right\} \cup \left\{s_i^{*(t)}\right\} \tag{4}$$

or equivalent matrix construction of $C_i^{(i+1)}$ with $c_i^{*(t)}$, $\tilde{C}_i^{(t)}$ and $C_i^{(i)}$. In words, we remove the retrieved premises and add the newly generated premise.

3.2.4. Construct Tree

Iteration stops once $|S_i^{(t+1)}| = 1$, i.e. we've arrived at the hypothesis. At this point we have $S_i^{(0)}, \cdots S_i^{(T_i)}$ for the total number of iterations T_i. Crucially, these can be use to reconstruct the tree y_i. The high-level approach of such an algorithm would be to construct the tree bottom-up from leaf nodes $S_i^{(0)}$, adding the generated premise nodes one by one.

With our model specified, let's continue with the implementation.

3.3. Variational Family

For the sake of computational tractability, variational inference is employed to approximate the posterior distribution $\mathbb{P}[c_{\text{test}}^{*(0)}, \tilde{c}_{\text{test}}^{(0)} \mid y, S_{\text{test}}^{(0)}]$. We denote the variational family $\mathcal{Q}$, where:

$$\begin{aligned} k_{\text{test}}^{(0)} - 1 &\sim \text{Binomial}(m_{\text{test}}^{(0)} - 1, \phi_1), \\ \tilde{C}_{\text{test}}^{(0)} &\sim \text{Retrieve}(phi_2, k_{\text{test}}^{(0)}, C_{\text{test}}^{(0)}), \\ c_{\text{test}}^{*(0)} \mid \tilde{C}_{\text{test}}^{(0)} &\sim \mathcal{N}(\phi_3 \bar{c}_{\text{test}}^{(0)}, I_d \phi_4), \end{aligned}$$

where

$$\begin{aligned} \phi &\triangleq (\phi_1, \phi_2 \phi_3, \phi_4) \in \Phi \\ &\triangleq [0,1] \times (\mathbb{R}^d \times \mathbb{R}^d) \times (\mathbb{R}^d \times \mathbb{R}^d) \times \mathbb{R}_+. \end{aligned}$$

4. Experimental Setup

4.1. Dataset

Our proposed model will be evaluated on the ENTAILMENTBANK dataset, containing 1 840 entailment trees, representing explanations for various science exam questions. The dataset has 5 881 total premises, with a mean nodes per tree of ≈ 7.6 (Ribeiro et al., 2022).

The ENTAILMENTBANK data considers 3 increasingly difficult tasks:

1. All initial premises are relevant,
2. Some initial premises are irrelevant,
3. The initial premises is the entire corpus.

To limit the scope of our experiment, we solely focus on Task 1.

4.2. Implementation

The source code for this experiment can be found at the following git repository. Selected source code has also been included in the Appendix.

4.2.1. DATA PREPROCESSING

The first implementation task is to preprocess the data. The data is in `jsonl` format, where each line is a `JSON` object corresponding to a single tree. Each `JSON` object includes a `lisp_proof` field, which describes the entailment tree structure as a lisp-style string, e.g. `((((((sent1 sent3) -> int1) sent2) -> int2) sent4) -> int3).` a recursive function `parse_trees` is provided to convert the plaintext into objects of type `EntailmentTree`, which is a custom type implemented by us. See {`entailmenttree, treenode, parsetree`}`.py` for reference.

4.2.2. SENTENCE EMBEDDING

Sentence encoding is performed with the pre-trained `all-MiniLM-L6-v2` sentence transformer with 384-dimensional embedding space. Initial training with the `paraphrase-distilroberta-base-v1` sentence transformer (768-dimensional) proved to be too slow on my hardware. Hong et al. also use a pre-trained transformer (2022) while Ribeiro et al. use the Siamese Network architecture from Reimers and Gurevych (2019). For performance reasons, the embedding dimension has been further reduced from 384 to 32 via PCA on a $N \times 384$ matrix of embeddings, where N is the total number of embeddings across every tree in the dataset. Initial trials were intractable on hardware used. See {`embed, parsetree`}`.py` for reference.

4.2.3. MODEL

Our probability model is implemented with PYRO, a probabilistic programming language (PPL) for PYTHON (Bingham et al., 2019). See `model.py` for reference.

4.2.4. GUIDE

To implement stochastic variational inference (SVI), a guide function is implemented to represent the selected variational family. See `guide.py` for reference.

5. Results

5.1. MCMC Methods

Attempting MCMC sampling with various configurations ($d \in 768, 384, 32$ with various sentence transformers and dimensionality techniques, dataset sizes $n \in \{1\,840, 1\,000, 500, 250, 100\}$) failed to compute in a reasonable amount of time (e.g. 1 day).

5.2. Variational Methods

Variational methods proved to be slightly more successful. With $d = 32$, $n = 1\,840$ and 500 iterations of SVI, we achieve the following ELBO loss:

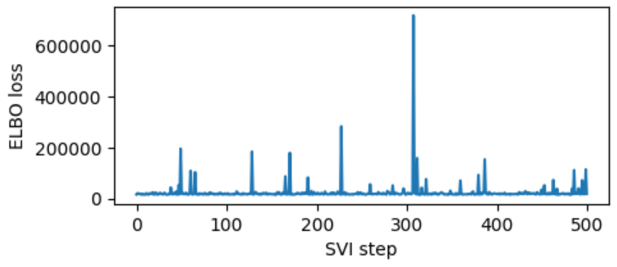

Figure 2. ELBO loss w.r.t. SVI step for 500 iterations

From the plot it is clear that ELBO loss remains relatively flat over time, indicating a lack of successful convergence. Furthermore, ELBO loss appears to be quite erratic, also indicating convergence issues, potentially to do with optimization.

6. Discussion

By far the greatest challenge faced in this experiment was the computational complexity of the project. Despite a seemingly small dataset of $1\,840$ entailment trees, initial experimentation with MCMC sampling methods (`pyro.infer.mcmc.nuts` (for Markov-Chain Monte Carlo (MCMC) via No U-Turn Sampling (NUTS)) proved to be intractable to compute on the hardware used.

Variational techniques proved to reduce the computation complexity of the experiment, yet it is clear that convergence has not been met. Possible reasons for this include:

1. Low embedding dimension ($d = 32$),
2. Model is not complex enough (e.g. $I_d\sigma^2$ variance for generation),
3. Guide (i.e. variational family) is not complex enough to allow for convergence in ELBO loss.

6.1. Limitations and Future Direction

To limit the scope of our experiment, we have made a few large assumptions. Firstly, our model assumes that all context are relevant to constructing the entailment tree. In practice, this is not necessarily the case, as other studies also consider an initial set of premises which include irrelevant premises.

A large bottleneck for the extent of this experiment was the lack of computation resources used for computing the model. The model was computed on a ThinkPad X230 without a dedicated GPU. Further experimentation on higher-performant hardware may allow for exploration of a wider range of model architectures. Furthermore, vectorizing the Pyro implementation of the model may improve performance, albeit difficult due to varying tensor sizes per sample and iteration. With increased performance, the following possibilities for exploration open up:

- Entailment graphs, not limited to entailment trees. E.g. Task 2 and Task 3, which include irrelevant premises.
- Various sentence embedding implementations. In our study we have only used one.
- Various priors. Our model only used a single set of priors. Further experimentation may be done with other prior specifications.
- Various parameterization improvements, e.g. heteroscedastic linear models with correlation.

While not a bottleneck for this particular experiment, future study may be limited by the small size of the ENTAILMENTBANK dataset of $5\,881$ total premises and $1\,840$ entailment trees. Especially considered relative to common embedding dimensions (e.g. $d = 384, 768$), the dataset is relatively sparse in embedding space, which may impede model performance.

7. Conclusion

In this study we propose a Bayesian approach to premise generation by constructing a probability model on entailment trees, iterative sampling an entailment tree bottom-up from initial premises i.e. leaf nodes. We expand on previous work by exploring premise retrieval premise generation through a Bayesian lens, defining priors and likelihoods on retrieved and generated premises. We implement the model with approximate variational methods in PYRO, a PPL for Python. Current exploration is limited by hardware and compute resource constraints, and a lack of data. There is room for optimization regarding model implementation to reduce computational complexity.

References

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: Deep universal probabilistic programming. *Journal of machine learning research*, 20(28):1–6, 2019.

Dalvi, B., Jansen, P., Tafjord, O., Xie, Z., Smith, H., Pipatanangkura, L., and Clark, P. Explaining answers with entailment trees. *arXiv preprint arXiv:2104.08661*, 2021.

DeYoung, J., Jain, S., Rajani, N. F., Lehman, E., Xiong, C., Socher, R., and Wallace, B. C. Eraser: A benchmark to evaluate rationalized nlp models. *arXiv preprint arXiv:1911.03429*, 2019.

Hong, R., Zhang, H., Yu, X., and Zhang, C. Metgen: A module-based entailment tree generation framework for answer explanation. *arXiv preprint arXiv:2205.02593*, 2022.

Jhamtani, H. and Clark, P. Learning to explain: Datasets and models for identifying valid reasoning chains in multihop question-answering. *arXiv preprint arXiv:2010.03274*, 2020.

Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*, 2020.

Le, Q. and Mikolov, T. Distributed representations of sentences and documents. In *International conference on machine learning*, pp. 1188–1196. PMLR, 2014.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

Ribeiro, D., Wang, S., Ma, X., Dong, R., Wei, X., Zhu, H., Chen, X., Huang, Z., Xu, P., Arnold, A., et al. Entailment tree explanations via iterative retrieval-generation reasoner. *arXiv preprint arXiv:2205.09224*, 2022.

A. Notation

Table 1. Notations

Sym.	Description
n	number of entailment trees
i	index on dataset; $i = 1, \cdots, n$
y	dataset of entailment trees
y_i	ith entailment tree
$S_i^{(0)}$	initial set of available premises for ith tree
t	current iteration
T_i	total number of iterations for tree i
$S_i^{(t)}$	available set of premises at iteration t
$m_i^{(t)}$	number of available premises at iteration t
$\tilde{S}_i^{(t)}$	retrieved set of premises at iteration t
$k_i^{(t)}$	number of retrieved premises at iteration t
$S_i^{*(t)}$	generated premise at iteration t
p	binomial parameter
W_r	linear parameter for retrieval
W_g	linear parameter for generation
σ^2	variance parameter for generation
$\theta_{il}^{(t)}$	categorical transformed parameter for retrieval
ϕ	sentence encoder
ψ	retrieval score function
$\mathcal{Q}$	variational family

B. Source Code

B.1. Entailment Tree Internal Model

```
import torch
from treenode import TreeNode
from embed import sentence_to_vec


class EntailmentTree:
  """
  Represents an entailment tree.

  Sentences and tree structure are decoupled.
  """

  def __init__(self, tree_json):
    # Sentences
    self.id_to_sentence = self._parse_sentences(tree_json)
    self.id_to_embedding =
    ↪  self._parse_embedding(self.id_to_sentence)

    # Tree structure
    tree_string = tree_json["meta"]["lisp_proof"]
    tokens = tree_string.replace('(', ' ( ').replace(')', ' )
    ↪  ').split()
    self.root = self._parse_root(tokens, 0, len(tokens) - 1)

  def __str__(self):
    return '\n'.join([f'{k}: {v}' for k, v in
    ↪  self.id_to_sentence.items()
                   ]) + '\n' + self.root.__str__()

  def generated_premises(self):
    """
    Returns a list of premises generated at each timestep
    based on the tree.

    :return: list of T ids, for T iterations.
    """

    def generated_helper(root):
      if len(root.children) == 0:
        return []

      # Recurse then concatenate
      ids = [generated_helper(child) for child in root.children]
      ids = [id for l in ids for id in l]
      ids.append(root.id)

      return ids

    return generated_helper(self.root)

  def available_premises(self):
    """
    Recursively fetches the IDs for the
    available premises at each iteration.

    :return: return list of T lists
    """

    def get_initial_premises(root):
      """
      Returns initial premises, a.k.a. leaf nodes.
      """
      if len(root.children) == 0:
        return [root.id]

      # Recurse and flatten
      ids = [get_initial_premises(child) for child in
      ↪  root.children]
      return [id for l in ids for id in l]

    ids_g = self.generated_premises()
    ids_r = self.retrieved_premises()

    # Construct available premises per iteration
    ids = [get_initial_premises(self.root)]
    for t in range(len(ids_g) - 1):
      ids.append(ids[-1].copy())

      # Remove retrieved premises
      for id in ids_r[t]:
        if id in ids[-1]:
          ids[-1].remove(id)

      # Add generated premise
      ids[-1].append(ids_g[t])

    return ids

  def retrieved_premises(self):
    """
    Recursively fetches the IDs for the
    retrieved premises at each iteration.

    The retrieved premises are simply a list
    of children for each node.
    Retrieved premises are ordered from left to right in the
↪   tree.

    Recursive algorithm:
    get list of retrieve premises for left, then right, then
    append (child for children of root) to result

    :return retrieved: list[T][K_t]
    """

    def retrieved_helper(root):
      if len(root.children) == 0:
        return None

      # Recurse
      ids = [retrieved_helper(child) for child in root.children]

      # Take out NoneTypes
      ids = [id for id in ids if id is not None]

      # Flatten
      ids = [id for l in ids for id in l]

      # Add current list of retrieved premises
      ids.append([child.id for child in root.children])

      return ids

    return retrieved_helper(self.root)

  def get_indices_of_retrieved_premises(self):
    """
    Returns the indices of each retrieved premise
    w.r.t. the available premises for each iteration.

    :return indices: T-long list,
    where T is the number of iterations.

    e.g. if ids_r = [[1, 4, 6], [1, 2]]
    and ids_a = [[1, 2, 3, 4, 5, 6], [0, 1 ,2]]

    then indices = [[0, 3, 5], [1, 2]]
    """

    ids_r = self.retrieved_premises()
    ids_a = self.available_premises()

    indices = []
    for r, a in zip(ids_r, ids_a):
      indices.append(torch.tensor([a.index(id) for id in r if
      ↪  id in a]))

    return indices

  def to_embedding(self, ids):
    """
    Given a list of m lists of m_i IDs,
    returns an list of m_i x d torch tensor of embeddings.

    The embedding for an ID is stored in the
    dictionary self.id_to_embedding

    e.g. self.id_to_embedding[id]
    returns a d-dimensional embedding.

    e.g. [[id1 id2] [id3 id4 id5]] -> [tensor1 tensor2]
    where tensor1 is 2 x d and tensor2 is 3 x d.
    """
    embeddings_list = []

    for id_list in ids:
      tensor_list = []

      for id in id_list:
        embedding = self.id_to_embedding[id]
        tensor_list.append(embedding)

      embeddings_list.append(torch.stack(tensor_list))

    return embeddings_list

  def _parse_sentences(self, tree_json):
    """
    Extracts sentences from tree json from dataset.
    Each sentences corresponds to an id.
    """
    return {
        **tree_json["meta"]["triples"],
        **tree_json["meta"]["intermediate_conclusions"],
    }

  def _parse_embedding(self, id_to_sentence):
    """
    Returns embeddings for each sentence.
    """
    return {
        id: sentence_to_vec(sentence) for id, sentence in
        ↪  id_to_sentence.items()
    }

  def _parse_root(self, tokens, start, end):
    """
    Must have a single root node.
    """
    id = tokens[end - 1]
    children = self._parse_children(tokens, start + 1, end - 3)
    root = TreeNode(id, children)

    return root

  def _parse_children(self, tokens, start, end):
    """
    Parses children into an array.
    """
    children = []

    i = start + 1

    while i < end:
      # Case 1: base case (leaf node)
      if tokens[i] != "(":
        id = tokens[i]
        children.append(TreeNode(id, []))
      # Case 2: recursive case
      else:
        j = self._find_matching_parenthesis(tokens, i)
        child = self._parse_root(tokens, i, j)
        children.append(child)
        i = j

      i += 1

    return children

  def _find_matching_parenthesis(self, tokens, left_index):
    left_count = 0
    for i in range(left_index, len(tokens)):
      if tokens[i] == '(':
        left_count += 1
      elif tokens[i] == ')':
        left_count -= 1
        if left_count == 0:
          return i
    return -1  # No matching right parenthesis found
```

```
class TreeNode:
  """
  Data structure representing tree structure.
  """

  def __init__(self, id, children):
    """
    Construct tree from lisp proof string.
    """
    self.id = id
    self.children = children

  def __str__(self):
    """
    Print tree.
    """

    return self._str_recursive(self, depth=0)

  def _str_recursive(self, node, depth):
    result = "  " * depth + node.id + "\n"
    for child in node.children:
      result += self._str_recursive(child, depth + 1)
    return result
```

B.2. Tree Preprocessing

```
import json
import pickle
import torch
import numpy as np
from sklearn.decomposition import PCA

from entailmenttree import EntailmentTree


def parse_trees(file_path):
  """
  Parses JSON encoded dateset stored at filepath
  and returns a list of trees.
  """
  trees_json = []
  with open(file_path, 'r') as file:
    for line in file:
      json_data = json.loads(line)
      trees_json.append(json_data)

  trees = [EntailmentTree(tree_json) for tree_json in trees_jso⌋
n]
  return trees


def reduce_embeddings(trees, d_new):
  """
  Given a list of trees,
  reduce the dimension of all embeddings
  from d to d' via PCA.

  :param trees: the list of trees
  :param d_new: reduced dimensionality

  :return trees: the updated list of trees
  """

  # Get N x d tensor of all embeddings
  embeddings = []
  for tree in trees:
    for id, embedding in tree.id_to_embedding.items():
      embeddings.append(embedding.unsqueeze(0))

  embeddings = torch.cat(embeddings, dim=0)
  print(f'embeddings.shape = {embeddings.shape}')

  # Fit PCA
  N, d = embeddings.shape
  pca = PCA(n_components=d_new)
  pca.fit(embeddings.numpy())  # Fit PCA on the data

  # Update embeddings
  for tree in trees:
    for id, embedding in tree.id_to_embedding.items():
      e = embedding.unsqueeze(0).numpy()
      tree.id_to_embedding[id] =
      ↪  torch.tensor(pca.transform(e)).squeeze()

  return trees


if __name__ == "__main__":
  dataset = "dev"
  original_dataset_fp = f'data/task_1/{dataset}.jsonl'
  processed_dataset_fp = f'data/processed/{dataset}.pkl'

  trees = parse_trees(original_dataset_fp)
  trees = reduce_embeddings(trees, 32)

  with open(processed_dataset_fp, 'wb') as file:
    pickle.dump(trees, file)
```

B.3. Pyro Model

```
import torch
import pyro
import pyro.distributions as dist

from embed import d


def model(Y, c_test):
  """
  Probability model for entailments trees.

  Inputs:
  Y:       input dataset of entailment trees.
  c_test:  input set of premises to sample a new premise from.

  Parameters:
  p:       p; binom parameter for k_i.
  wr:      parameters for linear premise retrieval.
  wg:      parameters for linear premise generation.
  c:       available premises.
  s2:      sigma squared.

  Transformed:
  c:       available premises.
  c_tilde: retrieved premises.
  c_star:  generated premises.
  n:       number of samples in dataset.

  Returns:
  c_tilde_test: retrieved premise.
  c_star_test: generated premise.

  Future work: vectorize implementation with pyro.plate.
  """
  n = len(Y)

  # Model parameter priors
  p, wr, wg, s2 = sample_model_params(d)

  for i in range(n):
    y = Y[i]

    # 1. Process data

    # Fetch available premises per iteration.
    c = y.to_embedding(y.available_premises())
    m = torch.tensor([c_t.size(0) for c_t in c])

    # Fetch retrieved premises per iteration.
    c_tilde = y.to_embedding(y.retrieved_premises())
    c_tilde_indices = y.get_indices_of_retrieved_premises()
    k = torch.tensor([c_tilde_t.size(0) for c_tilde_t in
    ↪  c_tilde])

    # Fetch generated premises per iteration.
    c_star = y.to_embedding([[id] for id in
    ↪  y.generated_premises()])
    # 2. Sample data

    # Sample for number of retrieved premises
    with pyro.plate(f"k_{i}", len(m)):
      pyro.sample(
          f"k_{i}^t",
          dist.Binomial(m - 1, p),
          obs=k - 1,
      )

    # Sample for retrieved premises
    theta = compute_theta(c, wr, k)

    # Construct theta for distinct sampling
    theta = [theta[t].repeat(1, k[t]) for t in range(len(theta) ⌋
)]
    for t in range(len(k)):  # iteration
      for j in range(k[t] - 1):  # sample number
        theta[t][c_tilde_indices[t][j], j + 1:] = 0

    for t in range(len(k)):  # iteration
      for j in range(k[t]):  # sample number
        # Sample then update theta
        pyro.sample(
            f"j_{i},{j}^({t})",
            dist.Categorical(theta[t][:, j]),
            obs=c_tilde_indices[t][j],
        )

    # Sample for generated premises
    Sigma = torch.eye(d) * s2
    for t in range(len(c_tilde)):  # iteration
      # Compute normal model parameters
      mean_c_tilde_t = torch.mean(c_tilde[t], dim=0,
      ↪  keepdim=True)
      mu_t = torch.matmul(wg, mean_c_tilde_t.T).squeeze()
      c_star_t = c_star[t].squeeze()

      pyro.sample(f"c_star,{i}^({t})",
                  dist.MultivariateNormal(mu_t, Sigma),
                  obs=c_star_t)

  # 3. Sample new premise
  m_test = c_test.shape[0]

  # Number of samples to consider
  # k_test = pyro.sample(
  #     "k_test",
  #     dist.Binomial(torch.tensor([m_test]), torch.tensor([p]) ⌋
),
  # ) + 1
  k_test = 2  # TEMPORARY

  # Retrieve samples
  theta_test = compute_theta([c_test], wr,
  ↪  torch.tensor([k_test]))[0].squeeze()
  j_test = []
  for j in range(k_test):
    # Sample then update theta
    idx = pyro.sample(f"j_test,{j}^(0)",
    ↪  dist.Categorical(theta_test))
    j_test.append(idx)
    theta_test[idx] = 0.0

  c_tilde_test = c_test[j_test, :]

  # Sample new premise
  mean_c_test_tilde = torch.mean(c_tilde_test, dim=0,
  ↪  keepdim=True)
  mu_t = torch.matmul(wg, mean_c_test_tilde.T).squeeze()
  c_star_test = pyro.sample(
      f"c_star_test",
      dist.MultivariateNormal(mu_t, Sigma),
  )

  return c_star_test, c_tilde_test


def sample_model_params(d):
  """
  Samples model parameters from corresponding priors.

  :param d: embedding dimension.

  :return p: binomial parameter.
  :return wr: normal parameter.
  :return wg: normal parameter.
  :return s2: normal parameter.
  """
  p = pyro.sample(
      "p",
      dist.Uniform(torch.tensor([0.0]), torch.tensor([1.0])),
  )

  with pyro.plate("wr", d * d):
    wr = pyro.sample("wr_entries",
                     dist.Normal(torch.tensor([0.0]),
                     ↪  torch.tensor([1.0])))
  wr = wr.reshape(d, d)

  with pyro.plate("wg", d * d):
    wg = pyro.sample("wg_entries",
                     dist.Normal(torch.tensor([0.0]),
                     ↪  torch.tensor([1.0])))
  wg = wg.reshape(d, d)

  s2 = pyro.sample(
      "sigmasquare",
      dist.Exponential(torch.tensor([1.0])),
  )

  return p, wr, wg, s2


def compute_theta(c, wr, k):
  """
  Construct evolved categorical distributions
  of retrieval probability for available premises.

  :param c: available premises, m x d tensor.
  :param wr: model parameters for retrieval, d x d tensor.
  :param k: number of distinct premises to sample.
  :param c_tilde_indices: indices of retrieved premises w.r.t.
  the available premises.

  :return theta: list of m_t x k_t tensors.
  """
  # Construct distribution on retrieved premises
  psis = [psi(c_t, wr) for c_t in c]
  psi_exp = [torch.exp(p) for p in psis]
  psi_exp_sum = [torch.sum(p_exp) for p_exp in psi_exp]

  theta = [psi_exp[t] / psi_exp_sum[t] for t in range(len(psis) ⌋
)]
  return theta


def psi(c, wr):
  """
  Scoring function to measure similarity of each
  available premise to mean available premise.

  :param c: available premises, m x d tensor.
  :param wr: model parameters for retrieval, d x d tensor.
  """
  mean_c = torch.mean(c, dim=0, keepdim=True)
  cTwr = torch.matmul(c, wr)
  scores = torch.matmul(cTwr, mean_c.T)

  return scores
```

B.4. Pyro Guide (Variational Family)

```
import torch
import pyro
import pyro.distributions as dist

from model import sample_model_params, compute_theta
from embed import d


def guide(Y, c_test):
  """
  Guide function to implement SVI.

  :param Y: trees (UNUSED).
  :param c_test: m_test x d tensor of initial premises.

  :return c_tilde_test: retrieved premise.
  :return c_star_test: generated premise.
  """
  p, wr, wg, s2 = sample_model_params(d)

  # Sample new premise
  m_test = c_test.shape[0]
  k_test = 2  # TEMPORARY

  # Retrieve samples
```

```
theta_test = compute_theta([c_test], wr,
↪  torch.tensor([k_test]))[0].squeeze()
j_test = []
for j in range(k_test):
  # Sample then update theta
  idx = pyro.sample(f"j_test,{j}^(0)",
  ↪  dist.Categorical(theta_test))
  j_test.append(idx)
  theta_test[idx] = 0.0

c_tilde_test = c_test[j_test, :]

# Sample new premise
mean_c_test_tilde = torch.mean(c_tilde_test, dim=0,
↪  keepdim=True)
mu_t = torch.matmul(wg, mean_c_test_tilde.T).squeeze()
Sigma = torch.eye(d) * s2
c_star_test = pyro.sample(
    f"c_star_test",
    dist.MultivariateNormal(mu_t, Sigma),
)

return c_star_test, c_tilde_test
```