Towards Neural Encoding of Higher-Order Cognition with Large Language Models

Sean Mata

Advisor: Professor Peter Ramadge, Professor Uri Hasson

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical and Computer Engineering
Princeton University

May 2025

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Sean Mata

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Sean Mata

Abstract

Large Language Models (LLMs) have demonstrated remarkable language understanding capabilities, significantly advancing neural encoding models. While recent progress has improved LLMs' reasoning abilities, they still face limitations in social contexts that require nuanced understanding of human cognition. This thesis investigates whether inference-time techniques like chain-of-thought reasoning—which externalize an LLM's reasoning process—can be leveraged for neural encoding of human reasoning and higher-order cognitive processing. Specifically, we focus on theory of mind and social reasoning within conversational contexts, examining the detection of subtext (unspoken beliefs, intentions, and desires) in communication.

Towards this we present two complementary projects: First, we develop a novel Theory of Mind-inspired Tree of Thought approach for dialogue generation that explicitly models subtextual reasoning. Our method produces diverse, contextually appropriate outputs, though automatic selection mechanisms currently struggle to outperform single-step generation. Through perturbation analysis and evaluator consistency tests, we gain insights into model confidence and the distribution of plausible responses. Second, we apply LLM-derived embeddings to neural encoding tasks, successfully modeling brain activity associated with both word- and sentence-level language processing using banded-ridge regression.Our findings suggest the potential for modifying sentence-level inputs to target the encoding of higher-order thinking in high-frequency Electrocorticography (ECoG) data. Together, these projects highlight both the promise and current limitations of LLMs in modeling social reasoning.

We conclude with proposals for future work, including improving training data, context structuring, evaluation metrics, and thought representation. Our findings point to new directions in aligning language models with human-like communicative competence and mental state inference.

Acknowledgements

An uncountable number of positive influences have shaped my journey to this point. For me, this thesis marks not just the culmination of a project, but also the nearing end of my undergraduate years—years that have defined much of my young adult life.

Within the context of this project, I am deeply grateful to Itamar Jalon for his mentorship and unwavering support. Without his guidance, I would have been lost in how to even begin tackling this work. Our regular meetings were often a highlight of my week, filled with discussions about neuroscience, as well as general life and the future. Thank you to Peter Ramadge for your invaluable support and for your guidance in managing both the research and the writing process. I am also especially grateful towards Professor Uri Hasson for welcoming me into his lab and facilitating the research opportunities that made this thesis possible. To all the members of the Hasson Lab, thank you for making me feel welcome, answering my many questions, and creating an environment in which I could learn and grow.

Beyond this project, there are many more people I wish to thank. At Princeton University, I am deeply grateful to everyone who helped me build a sense of home within this "orange bubble." I'm proud of the community we created—rich with stories, memories, and shared experiences. From travels across states and countries—New York, Chicago, Puerto Rico, Mexico, and more—to countless campus events spanning sports, music, and community and awareness-building, to the joy of watching and performing in dance shows, each moment added depth and color to my time here. I look forward to the friendships that will continue to grow long after graduation, and I hope to cross paths with many of these friends again in the years to come.

To my family: thank you for your constant love and support. I miss you every time I find myself living in a different city or country. To my parents, no matter where

I go, I will always be your baby. To my siblings, I will always be there to support you.

Finally, to my friends from Oxford, secondary school, and throughout the years: thank you for the countless moments of growth, learning, and laughter we shared. We navigated so many phases of life together, and I look forward to continuing to grow and experience new chapters with you and all my dear friends.

Contents

List of Figures

List of Symbols

$\mathcal{D} = \{D_1, D_2, \ldots, D_J\}$: Collection of dialogues, where each dialogue D_j is a sequence of utterances between two speakers.

$D = \{u_1, u_2, \ldots, u_N\}$: A dialogue between two speakers, represented as a sequence of N utterances.

$c_i = \{u_{i-w}, \ldots, u_{i-1}\}$: Context for utterance u_i, consisting of the w preceding utterances, or $\min(i-1, w)$ if $i - 1 < w$.

w: The number of preceding utterances included in the context for generating each response. This can be referred to as the context length or width.

$C = \{c_{w+1}, c_{w+2}, \ldots, c_N\}$: Set of all contexts used for response generation, corresponding to turns $i = w + 1$ to N.

$R = \{r_{w+1}, r_{w+2}, \ldots, r_N\}$: Model-generated responses for turns $i = w + 1$ to N, conditioned on context c_i.

$L = \{l_{w+1}, l_{w+2}, \ldots, l_N\}$: Human-written reference responses for turns $i = w + 1$ to N.

Note: For turns $i \leq w$, the model does not generate responses due to insufficient context.

S: Set of scores produced by the evaluation metric, $\{s_1, s_2, \ldots, s_N\}$.

Q: Set of human quality annotations, $\{q_1, q_2, \ldots, q_N\}$.

p_θ: A pre-trained language model (LM) with parameters θ.

$x, y, z, s, \ldots$: Language sequences, where $x = (x[1], \ldots, x[n])$ represents a collection of tokens, with each $x[i]$ being a token.

$p_\theta(x)$: The probability of sequence x under the pre-trained model p_θ, given by:

$$p_\theta(x) = \prod_{i=1}^{n} p_\theta(x[i] \mid x[1 \ldots i]).$$

$\bar{X}, \bar{Y}, \bar{Z}, \bar{S}, \ldots$: Collections of language sequences, represented in uppercase with a bar.

$p_\theta(y \mid \text{prompt}_{IO}(x))$: The probability of generating output y from input x wrapped in a prompt with task instructions and/or few-shot input-output examples.

$p_\theta^{\text{prompt}}(\text{output} \mid \text{input})$: Simplified notation for the probability of output given the input with a prompt.

prompt: A prompt that contains task instructions and/or few-shot input-output examples used to generate the next response.

$T = \{\tau_0^{(m)}\}_{m=1}^{M}$: The collection of subtexts $\tau_0^{(m)}$, indexed by m from 1 to M, where each $\tau_0^{(m)}$ represents a potential subtext idea based on a theory of mind taxonomy.

$G(p_\theta, s, k)$: Thought generator that generates k candidate thoughts based on the pre-trained language model p_θ and tree state s.

$s = [x, z_1, \ldots, z_i]$: A tree state, where x is the initial input, and $z_1, \ldots, z_i$ are the previous thoughts generated.

LLM- Large Language Model

ToM -Theory of Mind

Chapter 1

Introduction and Background

Large Language Models (LLMs) have recently exploded in popularity due to their ability to model language and their emerging, but imperfect, capacity for reasoning [2]. One application of LLMs is in human neuroscience, where they have been used to model neural signals from the literal content of words spoken [3]. This thesis aims to leverage recent advances in LLM inference-time reasoning to not only model the literal content of words during language production and comprehension but also subtext, meaning possible unspoken thoughts and inferences that arise during communication. Towards investigating this, this thesis presents two projects. The first applies an in-context learning technique, tree of thought, to a novel domain by aiming to improve dialogue generation over a naturalistic dataset by reasoning over potential subtextual thoughts — unspoken beliefs, desires, or intentions — inspired by theory of mind. The second builds on prior neural encoding projects in the Hasson lab, developing models that leverage both word-level and sentence-level embeddings to encode neural activity. The rest of this section will give a summary of related work and an outline of this report.

1.1 Reasoning versus language abilities in LLMs

It is under debate whether LLMs can truly reason in an abstract way to solve problems or whether they learn narrow-non-transferable skills for task-solving [4]. Nevertheless, they have had increasingly strong performance on reasoning benchmarks such as math, logic and reasoning [5] as well as language [6, 7]. This leads to the question of whether intelligence can be gained solely through mastery of language. This is related to a common fallacy conflating language and thought ability, where the ability to use language is neither a necessary nor sufficient condition for reasoning (Mahowald, Ivanova et al. coined the difference between knowledge of lingustic rules and understanding of language as formal linguistic competence and functional linguistic competence [8]). In humans, language and reasoning have been suggested to use different brain networks [9, 10], and individuals with aphasia who exhibit severe linguistic deficits still have other intact cognitive abilities such as social and mathematical reasoning [11]. Nevertheless, language can still bootstrap cognition, facilitating processes such as inner speech[12] and enabling faster development of social reasoning[13]. However, it seems unlikely that language models can acquire robust reasoning ability solely through next-token prediction during pre-training -motivating the development of various inference-time techniques aimed at enhancing LLM reasoning.

1.2 Inference time generation methods

Inference time generation methods are a method to improve the performance of LLMs without the large amounts compute and data needed for pre-training. This can be used alongside other methods of post-training improvement such as fine-tuning, reinforcement learning from human feedback and pure reinforcment learning in models such as o1 by OpenAI [14], R1 by DeepSeek [15], and QwQ by Qwen Team[16].

Inference-time methods come in many flavours, they can explore a single reasoning path often by optimizing a single prompt, such as in Chain of Thought [17, 18], multiple reasoning paths with or without multiple agents as in multi-agent debate [19] or boosting of thoughts [20] or can include an iterative or memory mechanism as in Reflexion [21] or buffer of thoughts[22]. However, there is no technique that consistently outperforms others in all tasks, so the implemented method must be chosen according to the specific task [23].

In this project, the strategy implemented is tree of thought [24]. This has the benefit of having multiple reasoning chains to track different ideas, as well as the flexibility to modify the tree's complexity and structure as needed for deeper or shallow levels of reasoning. Following an analysis of bias, a majority voting mechanism was also implemented [25].

1.3 Theory of mind in LLMs

Theory of Mind (ToM) refers to the cognitive capacity to infer the mental states—such as beliefs, desires, and intentions—of others, allowing us to predict and interpret their behavior. Traditionally, ToM has been modeled as a predictive model of others' actions based on their mental states. However, this approach often struggles with flexible generalization to novel situations and is limited in its use for complex planning tasks that involve many possible actions. A more powerful alternative frames ToM as a causal model—an abstract, structured representation of mental states that supports efficient reasoning, planning, and decision-making over a wide space of possibilities [26].

One domain where this richer model of ToM becomes essential is pragmatic language use. For instance, speakers frequently tailor their words to achieve specific emotional or social outcomes, such as using indirect or softened language to avoid offending someone. This type of communication requires balancing honesty, social norms, and an estimation of the listener's mental state. The Rational Speech Act (RSA) framework captures this complexity by modeling communication as a Bayesian reasoning process. Within RSA, speakers choose utterances by anticipating how listeners will interpret them, and listeners interpret utterances under the assumption that speakers are cooperative and informative. ToM is also central in interpersonal affect regulation, where individuals deliberately attempt to influence others' emotional states—calming them down, cheering them up, or helping them regain control. This kind of planning involves both empathy and strategic reasoning, highlighting the real-world complexity of applied ToM.

The extent to which large language models (LLMs) can perform ToM reasoning is the subject of ongoing debate [27, 28]. To address this, researchers have introduced a number of benchmarks to systematically evaluate ToM abilities in LLMs [29, 30, 31, 32]. While LLMs show strong results on certain tasks, their overall ToM performance remains inferior to that of humans, and is often accompanied by signs of overfitting [33].

Current datasets for evaluating Theory of Mind (ToM) in large language models (LLMs) suffer from notable limitations. Rich, human-authored narratives that naturally involve ToM reasoning are both rare and costly to curate. Consequently, researchers often rely on synthetic, template-based datasets that are overly simplistic and lack ecological validity. These datasets frequently include explicit mental state language (e.g., "Amy thinks that..."), which makes the reasoning task unrealistically easy and reduces the diagnostic value of the evaluation. Moreover, they tend to focus narrowly on action prediction, failing to capture the broader and more nuanced reasoning processes involved in real-world ToM. One dataset that attempts to address these shortcomings is SimpleToM [34], which includes both types of reasoning: the recognition of mental states (explicit ToM) and the application of that knowledge to predict actions or responses (applied ToM). They found that while LLMs tend to perform well on explicit ToM tasks, their performance on applied ToM remains weak—highlighting a critical gap and a central challenge relevant to our domain. Additionally, ToM performance in LLMs can vary significantly depending on the scenario, highlighting the need for methods that enhance LLMs' capacity for robust, context-sensitive reasoning rather than relying solely on narrow or superficial cues.

Several methods have been proposed to enhance Theory of Mind (ToM) performance at inference time [33, 35], but many of these approaches rely on few-shot examples or rigid assumptions that hinder scalability. While techniques such as custom chain-of-thought prompting can guide models toward more accurate ToM reasoning, they are often fragile and require extensive manual tuning [34]. Structured frameworks have also been introduced to support ToM reasoning. For instance, Wu et al. [36] developed COKE, a system that reasons over a knowledge graph by instantiating ToM as a collection of manually verified cognitive chains. These chains represent human mental activities in specific social contexts, including their behavioral and affective responses. More recently, Kim et al. [37] proposed a similar approach to the one in this project, propagating a series of hypothetical thoughts to explain and infer human actions with a method inspired by the sequential Monte Carlo algorithm.

1.4 Use of Language Models to study processing in the brain

A computational model's neural plausibility can be evaluated by assessing whether its internal representations align with patterns of brain activity elicited by corresponding stimuli. If a model consistently predicts neural responses to novel inputs, this suggests it captures key aspects of the brain's representational structure. This approach has been used in decoding brain activity in the visual pathway, using techniques such as motion energy models, Bayesian decoders, and deep learning frameworks [38]. Similarly, large language models (LLMs), which have achieved remarkable success in modeling human language, raise the question: how closely do their internal representations mirror those of the human brain?

To quantify representational similarity between LLMs and the brain, researchers typically employ two types of models: encoding models, which assess how well LLMs predict neural activity in response to linguistic stimuli, and decoding models, which infer the stimulus that produced a given neural response [39]. Empirical studies have revealed overlaps in representational structure—particularly between contextual embeddings from LLMs and decontextualized word embeddings. For example, fMRI studies [40] and intracranial recordings [41, 42] indicate that contextual embeddings better align with brain representations than static embeddings.

To deepen our understanding of this shared structure, Tuckute et al. [39] propose two key strategies for leveraging LLMs in the study of brain function. The first involves systematically probing LLMs by varying their architecture, training data, and behavioral outputs to isolate factors that align with brain activity. The second involves using LLMs as tools for in silico neuroscience experiments, allowing researchers to simulate and test hypotheses derived from empirical data.

These approaches have been extended to higher-level cognition. For instance, one study used fMRI to measure brain responses to 1,000 diverse sentences and showed that an encoding model based on GPT could predict the magnitude of the neural response to each sentence [43]. Interestingly, for sentences describing others' mental states, the model did not explain significantly more variance (beyond surprisal), suggesting that language-related brain regions are not modulated by social content. This finding supports the view that the brain's language network is functionally distinct from the theory of mind network.

One intriguing direction for future research is whether in-context learning—a method known to improve reasoning in LLMs—could be used to simulate or even enhance models of human reasoning. Such approaches may offer novel tools to investigate the mechanisms of cognitive processing in the brain.

1.4.1 Sentence embedding encoding

When working with fMRI signals, sentence-based encoding and decoding is more common due to the longer time duration of the signals. Early research demonstrated the ability to encode sentences using embeddings [44], and further analysis has shown that syntactic structure, rather than lexical-semantic content, is the primary contributor to the similarity between artificial neural network representations and brain responses in the language network [45]. Sentence embeddings are typically generated by averaging the word embeddings in the sentence or using the representation of the last sentence token (often the final period token ".") as a sequence summary.

1.5 Report Overview

In the following chapters, we will review the relevant literature and methodological background for natural language processing using transformers and dialogue modeling metrics, go over the methodology and discuss experimental results on the naturalistic 24/7 Conversations dataset, including future work and current limitations. Finally, we go over the method and results for the sentence encoding project and explore strategies for integrating it with the dialogue modeling work presented in this report.

Chapter 2

Methodological Background

2.1 Natural Language Processing using Deep Neural Networks

Natural language processing involves many different tasks which includes classification, text generation and sentence pair matching. A variety of deep neural network models can be used, including recurrent neural networks, long short-term memory networks, and most commonly now transformers. Modern transformer-based models based on the original architecture by Vaswani et al., 2017[1] give us the best performance on the relevant tasks for this project which requires text generation and sentence similarity scoring.

An overview of the transformer architecture is given in Figure 2.1. While a detailed account is covered in textbooks [46], the following section will give an overview of the key aspects of the models that we will repeatedly refer to.

Tokenisation

In NLP, a token is a single unit of text, such as a word, subword, or character, depending on the tokenization method used. For example, in the sentence "I love

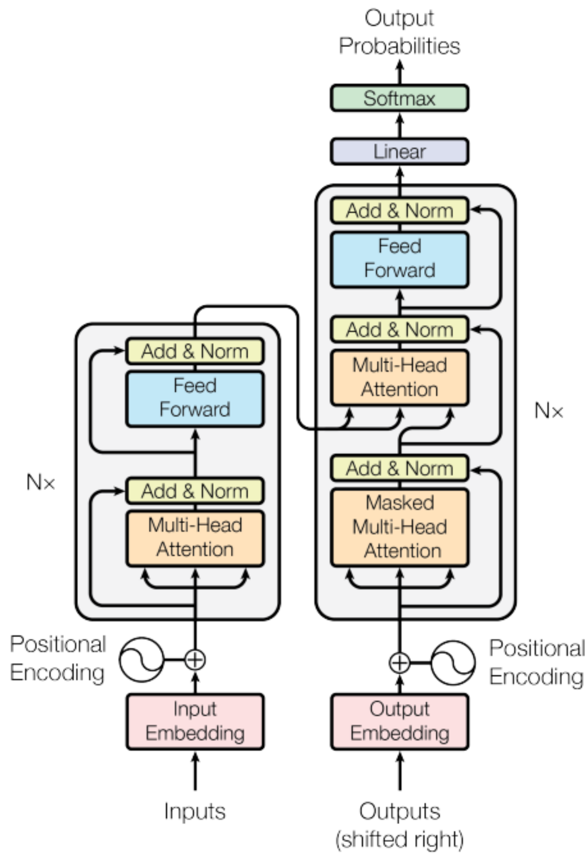

Figure 1: The Transformer - model architecture.

Figure 2.1: Transformer Architecture from Attention is All You Need (Vaswani et al., 2017)[1]

NLP!”, the tokens could be [”I”, ”love”, ”NLP”, ”!”] (word-level tokenization) or [”I”, ” lo”, ”ve”, ” NLP”, ”!”] (subword tokenization like in Byte Pair Encoding). Tokenization is a fundamental step in processing text for machine learning models. Different models may have different tokenization schemes with different vocabulary sizes and elements.

Static Embeddings

Static embeddings map each unique word in a vocabulary to a dense d-dimensional vector that capture meaning and similarity. This can be thought of as an improvement over one-hot encoding, where each word is represented as a large sparse vector with no meaningful relationship between each vector. In static embeddings, words with similar meanings have similar vectors and tend to cluster together in the embedding space. Furthermore, certain directions within this space can even capture specific relational meanings, for example, the vector difference between king and queen is similar to that between man and woman. The key attribute of static embeddings is that each word has a single fixed vector, regardless of its surrounding context. Common methods of creating these embeddings include Word2Vec and GloVe.

Contextual Embeddings

By contrast, with contextual embeddings, such as those learned by masked language models like BERT, each word w will be represented by a different vector each time it appears in a different context. The embeddings are made contextual through the self-attention mechanism.

The attention mechanism then measures how much focus each token should give to every other token. This is done by computing the similarity between the Query (Q) of a token and the Keys (K) of all tokens through the dot product QK^T. In the decoder only part this computation is masked, so can only look at the past tokens. The result is then scaled by $\frac{1}{\sqrt{d_K}}$ to normalize the values, helping stabilize gradients during training. Then the softmax function normalises the attention scores to ensure they sum to 1, converting raw similarity scores into a probability distribution. The result of this determines how much weight each token should assign to others in the sequence, and finally these computed attention weights are applied to the Value (V) vectors. Tokens with higher attention scores contribute more to the final representation of a given token. The weighted sum of values produces a new representation for each token, capturing contextual dependencies dynamically. This mechanism can be applied multiple times sequentially and for multiple heads. So when we refer to "contextual embeddings" this refers to weighted value vector after attention is applied using the Q and K matrices.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V \tag{2.1}$$

Next token prediction models

For generation of dialog I will use a decoder-only transformer e.g. DeepSeek, Mistral, GPT. In next-token prediction models like decoder-only transformers, text is generated one token at a time based on previous context. These models are also trained autoregressively. After processing input tokens, the hidden state of the last token in the final layer $h_N \in \mathbb{R}^{1\times d}$ is passed through a linear layer using an unembedding matrix $W \in \mathbb{R}^{d\times V}$ (called the language modeling head) to produce *logits* $z_i \in \mathbb{R}^{1\times V}$ — raw scores for each token in the vocabulary.

$$z_i = h_N W$$

These logits are converted to probabilities using the softmax function, producing $P(\text{next token} \mid \text{context})$:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}$$

The model can then either pick the most likely token (greedy decoding) or sample from this distribution (temperature sampling/ to generate the next token.

Masked word prediction

As opposed to autoregressive decoder models, masked language models like BERT operate by randomly masking words in sentences and training the model to predict these missing words using surrounding context. This approach falls under denoising training objectives, where inputs are deliberately corrupted through masking, substitution, reordering, deletion, or insertion, and the model learns to reconstruct the original input by minimizing cross-entropy loss between its predictions and the true values of the missing tokens.

This bidirectional approach enables learning rich contextual word representations applicable to various NLP tasks such as text classification, named entity recognition, and sentence similarity assessment. Often called encoder-only models, they excel at producing contextualized embeddings for interpretative tasks rather than generating text, making them particularly valuable for methods like BERTScore.

2.2 Text similarity metrics

For this project, we need a reference-based metric(s) to compare the similarity of the model generated response to the human reference response. In practise, human evaluation (typically from crowd-workers over experts for dialog) is used a gold standard for text and dialog evaluation with various strategies used for data collection and different questions asked to evaluators [47]. However, collecting this data from humans is time-intensive and expensive, so it is typically only used for a final evaluation, while automatic evaluation metrics are used during development to optimise model design and the choice of hyperparameters.

The ideal automatic metric maximally correlates with human judgement, however most metrics struggle to achieve this goal [48] hence metric-creation is still an open problem itself[49][50]. To improve accuracy or cost a hybrid approach may also be used, potentially combining multiple automatic metrics [51][52] or human and automatic metrics [53][54]. There have been several reviews on the several text similarity metrics [51] [55], and here we summarize their findings.

Formally, we can describe dialog evaluation metrics as follows: Given a dialog context $c = \{c_1, \ldots, c_m\}$, model response $r = \{r_1, \ldots, r_n\}$, and human-written reference response $l = \{l_1, \ldots, l_k\}$, the goal is to learn a function:

$$f : (c, r, l) \rightarrow s \tag{2.2}$$

that evaluates the generated response. (A metric is said to be reference-free if it can be expressed as a function of only c and r.)

These metrics are assessed by comparing them to human judgment. Concretely, a human annotator (or several annotators) scores the quality of a given response conditioned on the dialog context:

$$(c, r) \rightarrow q. \tag{2.3}$$

Given the scores produced by a particular metric, $S = \{s_1, s_2, \ldots, s_n\}$, and the corresponding human quality annotations, $Q = \{q_1, q_2, \ldots, q_n\}$, we can measure the performance of the metric by calculating the correlation between S and Q.

There are some important features and challenges to consider in the selection of this metric. The first is that the surface form representation is less important that the semantic meaning of the text, for example ""The cat sat on the mat.", "A mat was where the cat chose to sit.", ""The cat settled on the mat." , "The mat became the cat's resting place." should all receive similar similarity scores. Furthermore, the metric should be robust without changing greatly word small changes in word order or synonyms. Additionally, metrics can typically be evaluated at the utterance level as well as the system level by computing the average utterance-level score across many utterances [56], and having a strong correlation at the system-level is still useful even if the metric per utterance is noisy or has a lower correlation with human judgement.

Achieving these goals can be complicated, especially because of the long-tailed nature of language. You can not train a model on every possible combination of sentences because there are too many, which is why supervised trained metrics tend to overfit the dataset they were trained on [57]. Additionally, it is important to consider all metrics will have limiations and optimising a particular metric can lead to unintended effects [58][59].

In this section we go over the various options for evaluation of natural language text generation for dialogue, and explain the workings of BERTScore -the final metric selected. In the following sections, existing metrics can be broadly categorized into using n-gram matching, edit distance, embedding matching, or learned functions.

2.2.1 N-gram matching

N-gram matching metrics are based on word or character based overlap between the prediction and reference. The most popular example is BLEU, with other metrics including METEOR AND NIST. While originally used in translation to detect the textual similarity when changing language, they can also be used for evaluating textual similarity within the same language.

As an example, the BLEU score for a corpus of candidate translation sentences is a function of the n-gram word precision over all the sentences combined as well as a brevity penalty computed over the whole corpus [46]. Here n-gram refers to a sequence of token-length n. N-gram precision would be the percentage of n-gram tokens in the candidate translation that also occur in the regerence translation. The harmonic mean of the unigrams, bigrams, up to 4-grams is used in the calculation of BLEUscore. Since it is a word-based metric, it is very sensitive to word tokenization, making it impossible to compare different systems if they rely on different tokenization standards. It also cannot detect semantic equivalence, so two sentences with the same meaning but mostly different characters can receive an incorrect low score. Because of this, it is not generally used for sentence similiarity evaluation [60] and it is not a good fit for our uses.

2.2.2 Edit distance

Minimum edit distance/word error rate is another common metric with its origin in automatic speech recognition. However, since it is commonly used in speech recognition this equally penalises the completely wrong word in a sentence as well as a close synonym. Therefore, it is also misses capturing some of the idea of the semantic meaning of sentences. This lack of semantic meaning makes it unsuitable for this project.

2.2.3 Trained metrics

Various metrics are trained to optimise correlation with human judgement. The input to these models can include other different metrics such as character n-grams in BEER which used a regression model or ADEM which uses the context and the reference response as inputs to a recurrent neural network trained to predict appropriateness ratings by human judges. These methods usually have the highest correlation to human judgement [61]. However, these methods require expensive human judgments as supervision for each dataset and also generally have poor generalisation to new domains and data. These metrics have been shown to not be robust [57]. For example with ADEM, in 48.66% of cases the predicted score increased when the generated response was reversed. In 86.93% of cases the predicted score increased when the generated response was replaced with a dull dummy response.

2.2.4 Perplexity

Perplexity is a metric which was first introduced by Jelinek et al. in 1977 while working on automatic speech recognition [62], which was its original use case [63]. However, now it is most commonly now it is used to decide how to sample from a language model for text generation [64][65].

Given the probability assigned by the language model to a sequence $\mathbf{x}$ as:

$$p_\theta(\mathbf{x}) = \prod_{i=1}^{n} p_\theta(x[i] \mid x[1 \dots i-1])$$

Perplexity (PPL) can be derived from this probability as;

$$\text{PPL}(\mathbf{x}) = p_\theta(\mathbf{x})^{-\frac{1}{n}} = \exp\left(-\frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x[i] \mid x[1 \dots i-1])\right)$$

where n is the sequence length. Perplexity is highly interpretable, because it quantifies how "surprised" the model is by the given sequence—lower perplexity means the model assigns higher probability to the observed sequence. Perplexity can range from a theoretical value of 1, where the language model is completely certain of its prediction to infinity, where the language model is completely uncertain.

One example of its use in sampling is the maximum mutual information (MMI) scoring function [66] [67] has been used to predict such "boring" responses. MMI employs a pre-trained backward model to predict source sentences from given responses, i.e., $P(Source|target)$. The probability of$P(Source|Hypothesis)$can be used to rank the quality of hypotheses. Intuitively, maximizing backward model likelihood penalizes the bland hypotheses such as "that's interesting" or "ok", since the ability to discern the source sentence from such a target sentence is low, because these hypothesises can be associated with many possible queries. Another use case has been to evaluate the coherence of two consecutive sentences [68] for .

Perplexity is not as commonly used for evaluation of dialogue models. One reason for this is that this metric must be model-dependent, and it is better to avoid situations where the model generating the response also evaluates its quality because this would lead to bias [48]. Perplexity is also not comparable between language models with different vocabularies [63]. Moreover, perplexity is slightly different to the other metrics because it does not directly measure textual or semantic similarity between two samples. Even if two texts have similar perplexity values, they could be semantically very different. It could still be used to evaluate which subtext or thought (or no thought) most likely results in a given human reference, however it does not give us a direct measure of how well a model response matches the reference response. Hence, I did not prioritize implementing this metric.

2.2.5 Embedding based metrics

As discussed in Section 2.1, token (word) embeddings are learned dense token (word) representations that can be represented in a continuous vector space where semantic similarity is encoded so that tokens (words) with similar meanings are closer in the vector space. Embedding based metrics tend to have the highest correlation to human judgment based on their ability to match semantic meaning between candidate and reference sentences. Hence, this is what we chose to use in the rest of this report.

The general method of computing similarity is between two tokens is using the cosine similarity between two embedding vectors v and w as:

$$\text{cosine}(v, w) = \frac{v \cdot w}{\|v\| \|w\|}$$

As mentioned in pervious section, these embeddings can either be static and learned from large text corpora such as Word2Vec or GloVe, or they can be from contextual models like BERT. This is important for measuring semantic similarity of texts, since contextual embeddings allow the same word to have a different representation depending on its surrounding words, allowing the embeddings to capture nuanced meanings, polysemy, and relationships within the sentence context. For example, the word "bank" will have different embeddings in "river bank" versus "financial bank," reflecting its specific usage in each sentence. Static embeddings will measure the similarity of the word "bank" in these two sentences as exactly the same. Hence, using contextual embeddings for this task tends to do better[69].

The specific metric chosen for this project is BERTScore based on its superior performance for measuring textual similarity[70][69]. This metric is based on BERT, which is an encoder that creates contextual embeddings. Encoder-only models like BERT are known for producing high-quality, robust embeddings, making BERTScore an ideal choice for this task.

To explain how BERTScore works in detail, the BERTSCORE algorithm first passes the reference sentence x and the candidate sentence $\hat{x}$ through BERT, computing a BERT embedding for each token x_i and $\hat{x}_j$.

Given this sequence of tokens in a reference sentence $x = \langle x_1, \ldots, x_k \rangle$ and a candidate sentence $\hat{x} = \langle \hat{x}_1, \ldots, \hat{x}_l \rangle$, the similarity is calculated using a greedy matching algorithm.

Each token in x is matched to a token in $\tilde{x}$ to compute recall, and each token in $\tilde{x}$ is matched to a token in x to compute precision (with each token greedily matched to the most similar token in the corresponding sentence). BERTSCORE provides precision and recall (and hence F_1):

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\tilde{x}_j \in \tilde{x}} (x_i \cdot \tilde{x}_j)$$

$$P_{\text{BERT}} = \frac{1}{|\tilde{x}|} \sum_{\tilde{x}_j \in \tilde{x}} \max_{x_i \in x} (x_i \cdot \tilde{x}_j)$$

$$F_{\text{BERT}} = \frac{2 \cdot P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}$$

Additionally, BERTScore includes baseline scaling. From the formula mentioned above, scores would have the numerical range of cosine similarity between -1 and 1, however scores are observed in a smaller range in practice. To improve readability, BERTScore rescales its output with respect to an empricial lower bound b. This is computed by finding the average score between 1M candidate-reference pairs made by grouping two random sentences in the Common Crawl monolingual datasets. This lower bound is then used to rescale BERTScore linearly. The rescaled BERTScore is given as:

$$\hat{R}_{\text{BERT}} = \frac{R_{\text{BERT}} - b}{1 - b}$$

Chapter 3

Methodology

In this chapter, we will formalize the dialog generation task and describe the methods implemented to improve on this task. Then I will outline the generation strategies used to create a baseline score and the tree-of-thought and majority voting strategies implemented to aim to improve upon this score.

3.1 Problem setup

Given a dataset of conversational turns, we aim to construct a set of dialogues between two speakers. Each sample consists of a context, represented as a sequence of previous dialog turns $c_1, \ldots, c_m$, and a label, which is the true next utterance written by a human, denoted as $l_1, \ldots, l_k$.

The objective is to improve the ability of a pre-trained language model p_θ to generate a model response $r_1, \ldots, r_n$ that closely matches the human label response, $l_1, \ldots, l_k$. This similarity is measured using a metric such as BERTScore, which produces a set of scores $S = s_1, s_2, \ldots, s_n$ comparing the generated response to the human-written label.

Our approach is to enhance the model's generation process by conditioning its response on reasoning paths which are guided by prompts that consider a set of potential subtexts, ${\tau_0^{(i)}}_{c=1}^M$, which should help the model generate output that reflects the thought process the human speaker likely followed. By incorporating these reasoning paths into the model's response generation, we aim to produce more coherent and contextually appropriate responses that better match the human label. If successful, this would not only improves the quality of the model's output in natural dialogue settings but also demonstrates the model's ability to understand and reflect the underlying mental states involved in conversation.

3.2 Generation strategies

This work explores two distinct next-utterance generation approaches. The first is a straightforward baseline that conditions only based on previous context. The second seeks to improve upon this baseline by adopting a tree of thought reasoning framework, in which the model explicitly reflects on the next speaker's possible mental states and intentions before generating a response.

3.2.1 Single step utterance generation

This section describes our procedure for generating single-step utterances using large language models, and introduces the notation we use throughout. This notation based on the original Tree of Thought paper [24], with adaptations to suit the specific context of our work.

We begin by formalizing the general language generation process, before describing how we apply this framework to obtain a baseline generation score without any in-context learning. Let p_θ denote a pre-trained LM with parameters θ, and lowercase letters $x, y, z, s, \dots$ to denote a language sequence which is a collection of tokens, i.e., $x = (x[1], \dots, x[n])$ where each $x[i]$ is a token. Then $p_\theta(x) = \prod_{i=1}^{n} p_\theta(x[i] \mid x[1 \dots i])$. We also use uppercase letters with a bar $\bar{X}, \bar{Y}, \bar{Z}, \bar{S} \dots$ to denote a collection of language sequences.

We can turn a problem input x into an output y using a language model by wrapping input x with a prompt containing task instructions and/or few-shot input-output examples. We can write this as $y \sim p_\theta(y \mid \text{prompt}_{IO}(x))$. We can simplify this notation further by saying that $p_\theta^{\text{prompt}}(\text{output} \mid \text{input}) = p_\theta(\text{output} \mid \text{prompt}(\text{input}))$

Now we can describe the specific implementation approach. The relevant variables here are the context c and the prompt prompt . We can tie this together as:

$$r_1^{(i)} \sim p_\theta^{\text{prompt}} \left(r_1^{(i)} \mid c \right) \tag{3.1}$$

Figure 3.1 depicts what this generation process would look like where the context c consists of a single preceding utterance. We will create our baseline based on the generation procedure outlined in Equation 3.1. The prompt will be manually iterated to produce the best performance while the effect of changing context length will be included in the results.

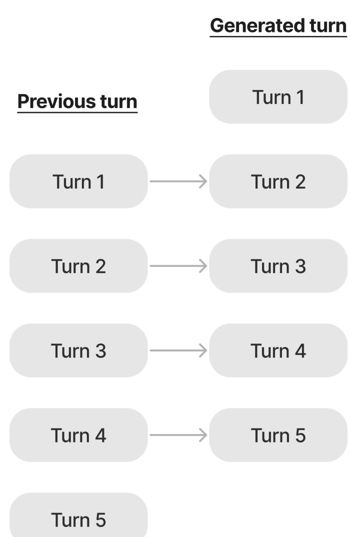

Figure 3.1: An illustration of the single-step generation procedure, where the model generates a response conditioned using a context length of 1 (only on the previous utterance in the conversation.)

3.2.2 Tree of thought

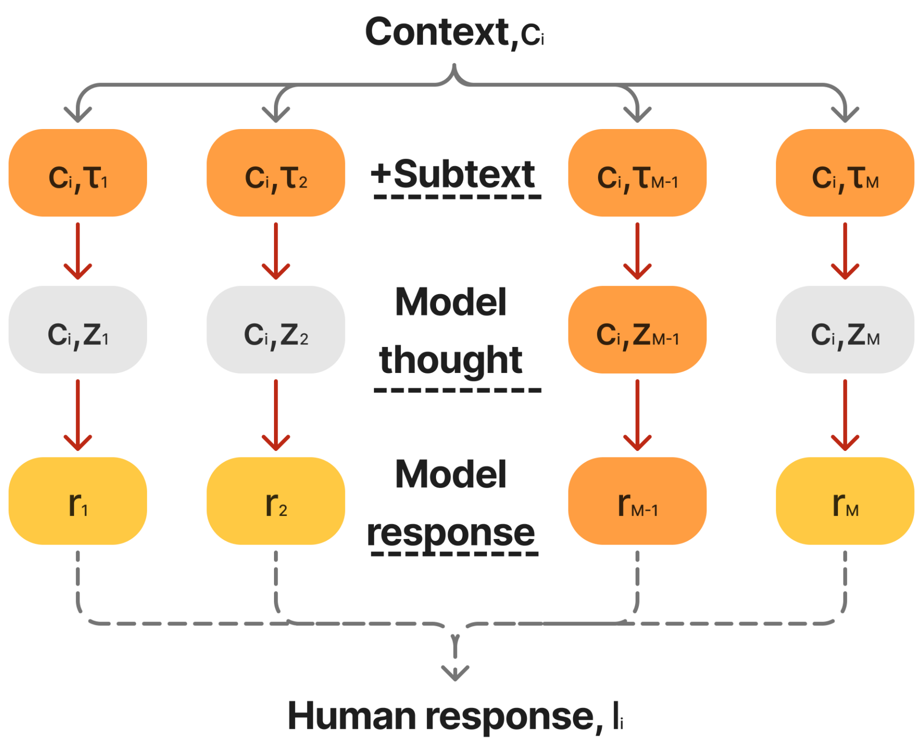

Figure 3.2: A diagram representing the Tree of Thought and evaluation methodology for utterance generation. Orange highlighted cells represent the actual states searched through using the LLM. Yellow highlighted cells represent the set of potential model responses. Red arrows represent LLM generation while the dashed arrows represent comparison using BERTScore.

In this work, we adapt the original Tree of Thought (ToT) framework to the domain of dialogue generation. To structure our method clearly, we follow the organization of the original ToT paper, which frames a specific instantiation of ToT around four key design questions: (1) How should the reasoning process be decomposed into intermediate thought steps? (2) How should potential thoughts be generated from each state? (3) How should states be heuristically evaluated? (4) What search algorithm should be used to navigate the tree? In the next section, we describe how we adapt and extend each of these components to address the challenges specific to dialogue generation.

At a high level, our method generates intermediate thoughts based on a fixed collection of subtexts of size M, which guide the generation of the final dialogue response. If all branches were explored, this would results in a tree of height 2 and width N as depicted in Figure 3.2, with the final layer corresponding to M candidate responses. However, in practice, the state evaluator $V(p_\theta, S)$ and search algorithm will be used to identify and follow the most promising branch without exploring the whole tree. All the explored states are highlighted in orange in Figure 3.2.

1. Thought Decomposition

While Chain of Thought samples thoughts without explicit decomposition, Tree of Thought designs and decomposes intermediate thought steps using the problem properties. In the following section, I will describe how the model uses a collection of subtexts $T = \{\tau_0^m\}_{m=1}^M$ to guide the reasoning process along specific paths. However, I will not focus on the decomposition of these thoughts, as we use a simple, low-depth tree structure, making further decomposition less necessary. This is because there is only one intermediate thought step before the model generates its response. Future work could explore increasing the depth of the tree to capture higher-order theory of mind or, alternatively, simplifying the thought process, as proposed in Chain of Draft [71].

2. Thought Generation

We require a thought generator $G(p_\theta, s, k)$ that generates k candidate thoughts based on the pre-trained language model p_θ and tree state s. Given a tree state $s = [x, z_1 \dots z_i]$, there are two strategies in the original paper to create a generator to create k candidates for the next thought step:

(a) **Propose thoughts sequentially using a "propose prompt"**:

$$[z^{(1)}, \dots, z^{(k)}] \sim p_\theta^{\text{propose}}(z_{i+1}^{(1\dots k)} | s).$$

(b) **Sample i.i.d. thoughts from a CoT prompt**:

$$z^{(j)} \sim p_\theta^{\text{CoT}}(z_{i+1}|s) = p_\theta^{\text{CoT}}(z_{i+1}|x, z_1 \dots z_i), \quad (j = 1 \dots k).$$

Proposing thoughts sequentially using a propose prompt works better when the thought space is more constrained. However, dialogue presents an inherently open thought space, suggesting that sampling i.i.d. thoughts from a CoT prompt would be more appropriate. The standard Tree of Thought approach would sample diverse reasoning paths independently like this, although sampling thoughts in this way may not lead to results that are interpretable from a theory of mind perspective.

To address these challenges, we instead initialize the first layer of tree states $S_0 = s_0^{(1\dots k)}$ by adding to the context c with a collection of subtexts $T = \{\tau_0^{(i)}\}_{i=1}^{M}$, which reflect different mental states or intentions the human speaker might have had. Specifically, each initial state is defined as $s_0^{(i)} = (c, \tau_0^{(i)})$. This can be interpreted as creating a manual thought generator $G_0(p_\theta, s_0 = c, k = M)$ which generates M new nodes given a context c based on the set of potential subtexts T, however this distinction is a matter of perspective rather than mechanism.

With the issue of creating thought diversity handled, the primary thought generator then does not need to calculate k new candidates given a tree state, instead it effectively sets $k = 1$ and is primarily concerned with propagating a node forward to produce intermediate thought(s) and then the final model response.

The final tree structure uses one intermediate thought which results in a tree structure of width M and height 2, which is summarized in Equation 3.2 and 3.3 below. Note that the prompts, and therefore the thought generators, for producing the intermediate thought and final model response are distinct because they create fundamentally different types of content. This contrasts with the original Tree of Thought framework, which does not explicitly distinguish between the generation of intermediate thoughts and final output, instead treating all nodes as being of the same type.

$$z_1^{(i)} \sim p_\theta^{\text{prompt1}} \left(z_1^{(i)} \mid \tau^{(i)}, c \right) = G_1(p_\theta, s_0^{(i)}, 1) \tag{3.2}$$

$$r^{(i)} = z_2^{(i)} \sim p_\theta^{\text{prompt2}} \left(r_2^{(i)} \mid z_1^{(i)}, \tau^{(i)}, c \right) = G_2(p_\theta, s_1, 1) \tag{3.3}$$

In this implementation discussed in the results, a set of six subtexts are used, based on a framework of self versus other for feelings, goals and information seeking within a larger taxonomy of theory of mind reasoning [72].The selection of these subtexts are discussed in more detail in Section 1. The exact prompts are specified in Appendix XYZ.

3. State Evaluator

Given a series of different thoughts we also need a state evaluator $V(p_\theta, S)$ to evaluate the progress they make towards solving the problem. Traditionally, this is used as a heuristic for a search algorithm to determine which states to keep exploring and in which order. We use the language model to reason over these states/thoughts to choose which one to generate a response for. I implemented both strategies that the original paper provides to evaluate states either independently or together:

(a) **Value each state independently:**

$$V(p_\theta, S)(s) \sim p_\theta^{\text{value}}(v|s), \quad \forall s \in S,$$

where a value prompt reasons about the state s to generate a scalar value v (e.g., 1-10) or a classification (e.g., sure/likely/impossible) that could be heuristically turned into a value.

(b) **Vote across states:**

$$V(p_\theta, S)(s) = \mathbb{1}[s = s^*],$$

where the "best" state $s^* \sim p_\theta^{\text{vote}}(s^*|S)$ is voted for based on deliberately comparing different states in S using a vote prompt. This is useful when problem success is harder to directly value.

We additionally define two reference state evaluators for comparison. The first is the random evaluator which can act as a baseline and the second is a perfect (oracle) evaluator which would represent an ideal upper bound.

(c) **Random state evaluator:**

$$V_{\text{random}}(S)(s) = \frac{1}{|S|}, \quad \forall s \in S,$$

where each state $s \in S$ is selected uniformly at random. This gives an equal value to each state, so that one is selected randomly at each level of the tree.

(d) **Perfect state evaluator:**

$$V_{\text{perfect}}(S)(s) = \mathbb{1}[s = s^*],$$

where:

$$s^* = \arg\max_{s \in S} \text{BERTScore}(\text{complete}(s))$$

and complete(s) denotes the best possible full solution reachable from state s.

This always chooses the state that will end up maximising the BERTScore. While defining this function is very hard (and using LLMs to approximate this is a focus of this thesis), by propagating each branch of the tree forward to completion and backtracking we can deduce what the ideal state evaluator would have chosen.

4. Search Algorithm

Finally, within the Tree of Thoughts (ToT) framework, one can use different search algorithms depending on the tree structure. The original paper mentions the depth first and breadth first algorithms as starting points. Given the simple tree structure here, one of the main constraints is the accuracy of the state evaluator $V(p_\theta, S)$. In particular, it is likely easier for the language model to evaluate the likelihood of a particular mental state directly using a subtext instead of a thought generated by that subtext, because it will potentially match what it has seen in its training distribution better.

Hence, the final algorithm we implement can be thought of a breadth first search over the first layer, including (context+subtext) with a greedy rollout of the most promising state to produce a thought based on the subtext and then the final model response as in Equations 3.2 and 3.3.

The algorithmic implementation of this for this tree structure is included below.

Algorithm 1 Greedy Rollout After Breadth-First Search

Require: Input x, LM p_θ, subtexts $T = \{\tau_0^{(i)}\}_{i=1}^N$, thought generator $G()$, state evaluator $V()$.

1: $S'_0 \leftarrow \{[x, \tau_0^{(i)}] \mid \tau_0^{(i)} \in T\}$
2: $V_0 \leftarrow V(p_\theta, S'_0)$
3: $s_0^* \leftarrow \arg\max_{s \in S'_0} V_0(s)$
4: $s_1 \leftarrow G_1(p_\theta, s_0^*, 1)$
5: **return** $G_2(p_\theta, s_1, 1)$

3.3 Ranking and State Evaluation Metrics

For being able to characterise the ranking of the thoughts as well as the consistency of the voting process, there are various metrics we will use.

Ranking thoughts

Average Reciprocal Rank

To be able to rank the thoughts we use the Average Reciprocal Rank (ARR) which is very similar to the Mean Reciprocal Rank used to evaluate the effectiveness of ranking systems such as search engines and recommendation systems. The main difference is that ARR is calculated per thought instead of per-query. It is given as:

$$ARR_i = \frac{1}{N} \sum_{k=1}^{N} \frac{1}{r_{ik}} \tag{3.4}$$

Where ARR_i is the Average Reciprocal Rank of thought i and r_{ik} is the rank of thought i in trial k. This creates a score that varies from worst at $1/N$ to a best score of 1. While doing a sum of ranks would treat 1st vs 2nd vs 3rd as linear differences, MRR or calculating the number of times a thought is ranked first would only consider the rank of the 1st ranked thought. Hence, ARR provides a middle ground where being 1st is prioritized, but having a good ranking such as always ranking second will still have a good score.

Measuring consistency in voting

When prompting the LLM with the subtext thoughts in different orders, I observed that the voted thought depended on the order of thoughts in the prompt. Ideally, the selection mechanism would be perfectly consistent, however there must be some bias or uncertainty in the voting process that decreases this consistency. We can quantify the consistency of the voting process using two complementary measures:

1. Maximum Agreement Score

For each sample we define $V_{i,j} = \{v_{i,j}^{(1)}, v_{i,j}^{(2)}, \ldots, v_{i,j}^{(L)}\}$ as the set of votes received by thought j in sample i across L shuffles, where: $v_{i,j}^{(\ell)} = \begin{cases} 1 & \text{if thought } j \text{ was selected in shuffle } \ell, \\ 0 & \text{otherwise.} \end{cases}$

Then the count matrix $C_{i,j}$ represents the total number of times thought j was selected for sample i across all shuffles:

$$C_{i,j} = \sum_{\ell=1}^{L} v_{i,j}^{(\ell)}$$

(Note that the number of shuffles can be recovered by calculating $L = \sum_{i=1}^{k} c_i.$). The maximum agreement score for sample i is then:

$$\max_j C_{i,j}.$$

This score ranges from T/L (minimal consistency) to L (perfect consistency). Higher values indicate stronger robustness to ordering effects.

2. Evenness Index

The evenness index provides a normalized measure of vote distribution uniformity for sample i. It ranges from 0 to 1, where a value of 1 indicates the distribution is uniform and highly inconsistent, while a value of 0 represents perfect consistency. It is defined as:

$$E_i = \frac{H_i}{\log_2(L)} = \frac{-\sum_{i=1}^{k} p_i \log_2 p_i}{\log_2(L)} \tag{3.5}$$

Where E_i is the evenness index for sample i, H_i is the Shannon entropy of the vote distribution at sample i and $p_j = \frac{C_{i,j}}{L}$ is the probability that thought j was selected at sample i. The denominator $\log_2(L)$ normalizes the entropy by the maximum possible entropy over L shuffles, providing a measure of how evenly the votes are distributed across the thoughts.

3.4 Majority Voting

Using the measure of consensus described in the previous section, we can create a quasi-mixture of experts voting approach by aggregating scores/choices over different order of subtexts and selecting the most popular vote.

Effectively, this creates a meta-state evaluator, that takes in the output of the state evaluator $V(p_\theta, S)(s)$ over L shuffles. The voted subtext thought for sample i using count matrix $C_{i,j}$ by seeing what the most popular thought is would be:

$$\arg\max_j C_{i,j}.$$

Chapter 4

Experimental Results

4.1 Experimental Setup

4.1.1 Dataset curation

The first step of the experimental setup is setting up the dataset. For a general dialogue dataset with a series of utterances between two speakers, the first step is to segment it into a collection of dialogues, where each dialogue $D = \{u_1, u_2, \ldots, u_N\}$ is represented as a sequence of N utterances between two speakers. After each dialogue is created, we can extract a series of contexts $C = \{c_{w+1}, c_{w+2}, \ldots, c_N\}$ which are used to generate model responses $R = \{r_{w+1}, r_{w+2}, \ldots, r_N\}$ to compare to human reference responses $L = \{l_{w+1}, l_{w+2}, \ldots, l_N$.

To construct the sets C and L, a sliding window of size w is applied over each dialogue. For each turn i from $w + 1$ to N, we collect the w utterances immediately preceding u_i as the context c_i, and define the corresponding label l_i as the ground-truth utterance u_i. Since at least w previous utterances are required to form a complete context, the model only generates responses for turns where $i > w$. This ensures that predictions are all conditioned on the same amount of prior context for fairness.

24/7 Conversations

To implement this setup for the "24/7 Conversations" dataset, a series of preprocessing steps are required. To give some context, the "24/7 Conversations" dataset contains 750 hrs of continuous electrocorticography (ECoG) data from epilepsy patients engaging in free daily life conversations alongside audio recordings using a microphone to capture all the speakers in the room. These recordings have already been preprocessed by members of the Hasson lab using an Automatic Speech Recognition system to generate a time-stamped transcript. For one of the patients, patient 798, we have this data segmented by sentence with speaker labels.

To prepare the dataset as a collection of dialogues $\mathcal{D}$, we first transform the transcript into a sequence of utterances. Multiple consecutive sentences spoken by the same speaker are merged into a single utterance. After constructing the collection of dialogues $\mathcal{D} = \{D_1, D_2, \ldots, D_J\}$, we apply steps which aim to filter out interactions that lack meaningful turn-taking and exchange or are not useful for modeling conversational dynamics.

The first filtering step targets one-sided conversations, which are identified based on the ratio of speaker turns. Specifically, we exclude any dialogue in which a single speaker contributes more than 75% of the utterances, as these are less likely to have meaningful two-way exchanges. This step is visualized in Figure 4.1. In this dataset, we observe several instances where one participant—typically a doctor—dominates the conversation, often explaining something while the other speaker listens passively. These are some of the interactions we specifically seek to exclude.

Finally, we apply a length-based filter to the reference responses. Many utterances in naturalistic dialogue are extremely short—often backchanneling cues such as "mhm," "yeah," or "okay"—which are challenging to model and offer limited value for evaluating response generation. On the other hand, extremely long utterances introduce their own modeling difficulties. Therefore, in our analysis, we restrict at-

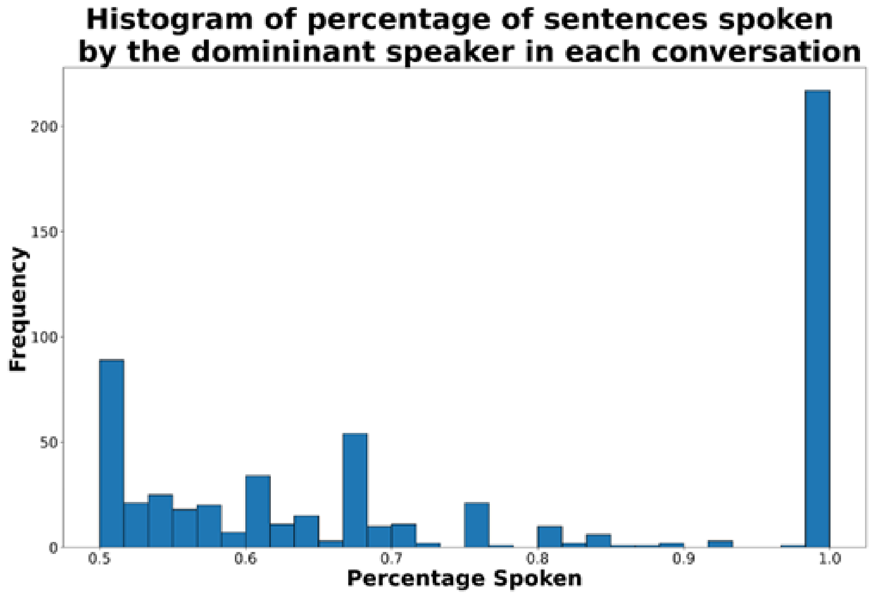

Figure 4.1: A histogram of the percentage of sentences spoken by the dominant speaker in each conversation

tention to responses containing between 5 and 25 words. Then, for patient 798, the final dataset consists of 827 labels across 318 dialogues, the average number of turns per dialogue being 13.3. These statistics correspond to the case where the maximum number of context turns used in the analysis is $w = 15$. A smaller value of w would yield more datapoints, as fewer turns would be excluded for lacking sufficient preceding context.

(The code for this filtering is included in Appendix A.3)

4.1.2 Large Language Model Configuration

When selecting a language model for this project, we considered various models with different trade-offs, such as ease of use, performance on specific benchmarks, and flexibility for future fine-tuning. After careful evaluation, we chose the Llama 3.x series of models [73], which offers several key advantages. Notably, at the time of development it was one of the most powerful models with publicly available weights, providing the flexibility for potential future fine-tuning. Additionally, Llama 3.x is available in multiple sizes—1B, 7B, and 80B parameters—trained on similar datasets, providing both adaptability across different use cases and the ability to assess the impact of model size on performance.

In particular, we use the instruction-tuned versions of Llama 3, which are specifically fine-tuned to improve the quality of generations by understanding more complex requests and producing more relevant responses. Generally, models are fine-tuned for specific tasks such as question-answering, summarization, and coding. Without this fine-tuning, the model would only predict the next token based on statistical patterns in the training data, leading to less rich and context-aware dialogue. The instruction-tuned models also enable us to incorporate a "system prompt," which helps shape the style and content of the model's responses.

The hyperparameters, such as temperature, that control the sampling behavior for generating the next token, are detailed in Appendix A.5.

4.2 One step utterance generation

This section will describe the results of the single step utterance generation method described in Section 3.2.1. The system prompt for this generation is given in Appendix A.1. The code for generation is given in Appendix A.4.

4.2.1 Sentence generation improves with model size and context length-Up to a Limit

Figure 4.2 presents the average BERTScore across all samples, varying by model size and context length (measured as the number of preceding utterances). Broadly, we observe that BERTScore improves with both increasing model size and longer context. However, the gains from additional context diminish beyond approximately 10 preceding turns. This could suggest that speakers typically do not reference dialogue more than 10 turns prior as often, or that excessive context may dilute the model's attention to relevant parts of the conversation.

With regard to model size, both the 8B and 70B parameter models outperform the 1B model as context increases, with the 70B model slightly edging out the 8B.

This score is computed by averaging the BERTScore across randomly shuffled candidate-reference label pairs, which approximates the baseline generation method used in the original BERTScore paper [69] (the primary difference being that the original paper compares 1M pairs).

Two baseline scores are included as dashed lines in Figure 4.2. The red line represents the average BERTScore when comparing each human label l_a to another randomly selected label l_b from the dataset. Computing the score over randomly shuffled candidate- reference pairs of labels in this way approximates the method used to create a baseline in the original BERTScore paper [69] (the main difference being that the original paper compares 1M pairs). The blue line shows the average BERTScore between human labels and random generations from an 8B model without context. These baselines offer reference points: for instance, we expect models to perform no worse than the random generation baseline of 0.638.

Interestingly, the 8B and 70B models only begin to surpass the pairwise random-label baseline after a context length of 5, while the 1B model never exceeds it.

To evaluate the effectiveness of in-context prompting methods in later sections, we adopt the BERTScore of the 8B model with 10 turns of context as our model baseline because this configuration yielded the highest observed score.

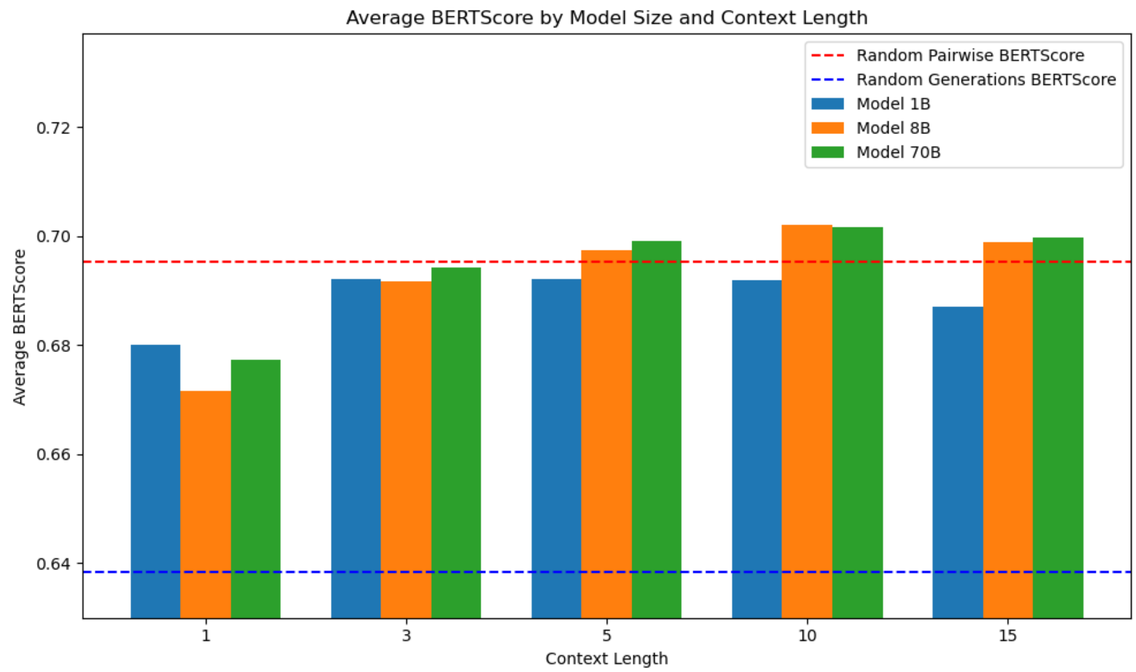

Figure 4.2: Average sentence similarity scores between model generated and human reference turns of conversation for differing amounts of context and different model sizes

4.3 Thought generation for theory of mind reasoning

This section will go over the results of the multi-step sentence generation based on the thought generator by evaluating all possible branches in the tree of thought. By characterizing the performance of each branch, we can characterize the problem of choosing the best thought in more detail. Overall, I find that the model response based on the best subtext outperforms the baseline, however randomly selecting a subtext to generate based off of will not improve the baseline, hence being able to find the best ranked thought is important.

4.3.1 The best generated responses based on subtext thoughts improve upon baseline

Figure 4.3 presents the distribution of scores for each ranked thought, with thoughts ordered from highest to lowest score within each sample. The box plot illustrates how the top-ranked, second-ranked, third-ranked, and subsequent thoughts perform across all samples. Notably, selecting the model response associated with the first or second highest-ranked subtext consistently outperforms the baseline. In contrast, the third-ranked thought performs comparably to the baseline on average, while lower-ranked thoughts generally underperform. These results highlight a clear opportunity to improve upon the baseline by selecting higher-ranked generations.

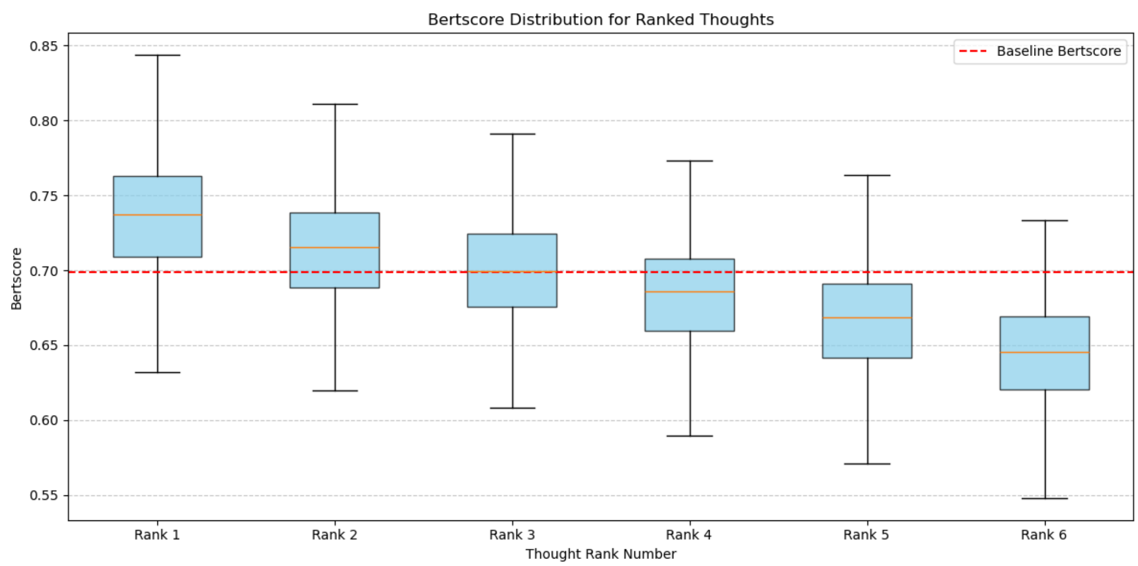

Figure 4.3: Box plot showing the BERTScores for all ranked ToM subtext generations (1st, 2nd, 3rd best, etc.) for each sample.

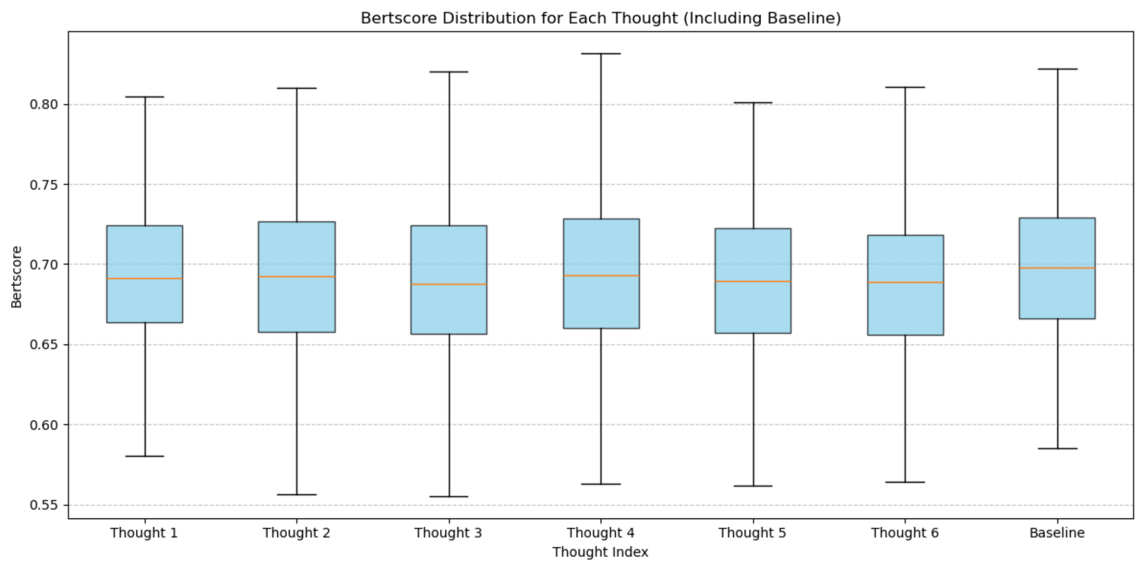

Figure 4.4: Box plot showing the BERTScore distribution for each different subtext over all samples

4.3.2 Selecting Responses Based on a Single Subtext Underperforms Baseline

While choosing a highly ranked generation can outperform the baseline, consistently selecting responses based on a single subtext tends to slightly underperform. This trend is evident in Figure 4.4, which presents the average BERTScore for model responses categorized by subtext. From a Theory of Mind perspective, this result is logical: applying the same inferred mental state across all interactions overlooks the nuances of individual interactions and lacks conversational specificity to the context. As a result, biasing the selection mechanism toward any single subtext would not necessarily lead to any improved performance.

4.3.3 Self-Reflective and Goal-Oriented Thoughts Rank Higher

We use the Average Reciprocal Rank (ARR), defined in Equation 3.4, to evaluate the relative ranking performance of different subtexts. As shown in Figure **??**, there is some variation across subtexts, though none deviate drastically from the expected

ARR for a uniform distribution of ranks, given by $\frac{1}{M}\sum_{k=1}^{M}\frac{1}{k} = 0.408$. Notably, subtexts that prompt the model to think about *goals* tend to achieve higher ARR scores, while those focused on *decision-making* or *information processing* perform slightly worse. Across all subtexts, prompting the LLM to consider the *upcoming* speaker's state of mind from their own perspective generally yields better rankings than prompting it to reflect on the *previous* speaker's state.

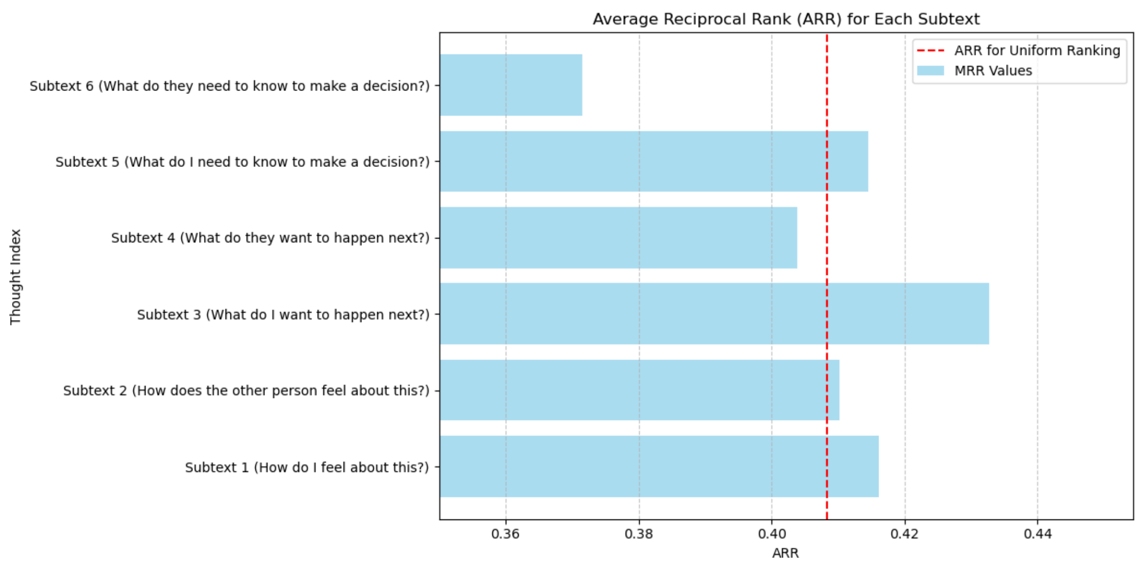

Figure 4.5: Bar chart showing the Average Reciprocal Rank for each theory of mind subtext. The possible range is between 0 and 1, with a perfect score of 1 being achieved by having a rank of 1 over all samples

4.3.4 Sampling multiple times from a varied distribution can still lead to improvement in BERTScore

Having shown that a thought generation approach based on a set of subtexts can improve text similarity scores, we next explore how perturbing this method affects performance.

Figure 4.6 summarises this analysis. To produce this graph, we perturb several key components involved in dialogue generation—the context c, the list of subtext

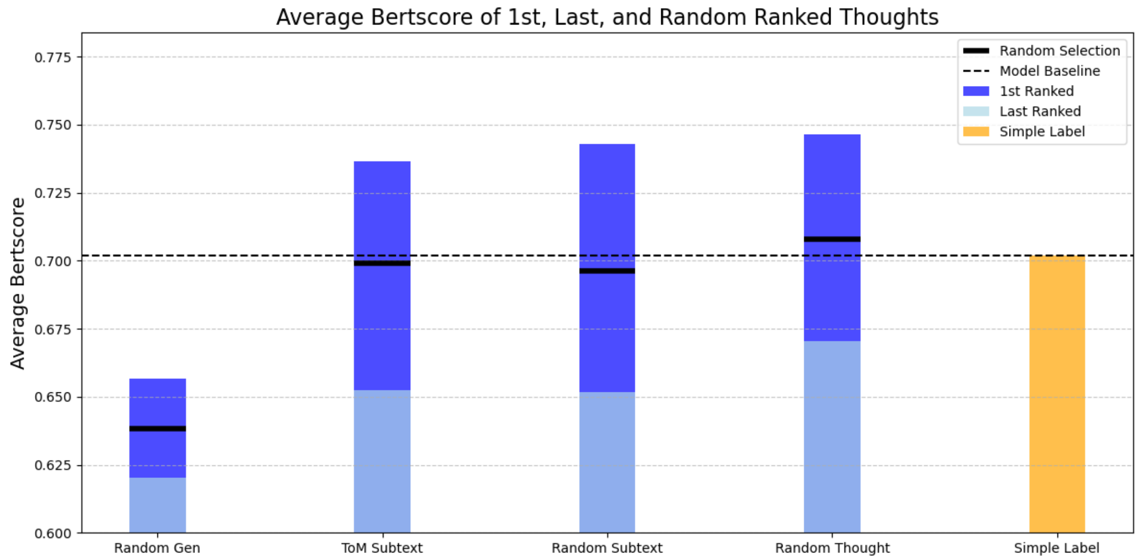

Figure 4.6: BERTScore outcomes for different generation strategies with various perturbation conditions. Dark blue bars represent the average best score per sample, light blue bars the worst and thick black bars the score from randomly sampling a subtext each sample. The yellow bar shows the average score with a simple label. The black dashed line is the model baseline.

thoughts T, and the intermediate generated thought z. The only major variable that remains unchanged is the prompt.

To perturb the context c, we use the Random Generation (RandomGen) condition. As in Section 4.2, this involves removing the context from the LLM input and instead sampling random text generations. We sample $M = 6$ generations to match the multi-sample setup used when conditioning on different subtexts.

For the Random Subtext condition, we replace the theory-of-mind-based subtexts with a set of unrelated, casual thoughts that are not clearly tied to mental states or intentions. The set includes: "Hm. Where was I going?" "Mmm. I could have toast later." "I need to remember to call my mom later." "What's the weather like today?" "I need to remember to pick up groceries later." "Do fish ever get thirsty?"

In the Random Thought condition, we retain the original theory-of-mind subtexts, but instead of generating thoughts conditioned on the subtext and context, we substitute the intermediate thought z with a completely random text sample.

We also include a simple baseline label—"Yeah, that's what I see too."—as a final comparison. Since this case does not involve multiple generations, it is plotted as a single yellow bar in Figure 4.6.

The figure shows the following: the light blue bars represent the average worst score across samples for each condition, while the dark blue bars show the average best score. The thick black horizontal lines indicates the expected score if a sample were selected at random. Lastly, the dashed black line represents the baseline model response from Section 4.2.

First, note that sampling multiple times from a distribution with some variance and choosing the best outcome will, by definition, outperform the mean of the distribution. This section evaluates that intuition in practice by modifying the generation process to inject variety.

The results confirm that all generation strategies exhibit variance in output quality, meaning that sampling multiple times and choosing the best result consistently improves BERTScore. Interestingly, both the theory-of-mind and random subtext conditions yield similar distributions, with the random subtext condition exhibiting slightly more variance—resulting in a higher best-case score.

Surprisingly, the Random Thought condition outperforms both subtext-based strategies. One plausible explanation emerges from examining the performance of the Simple Label condition, which closely matches the model baseline despite its simplicity. This suggests that basic or generic statements such as "Yeah, that's what I see too" can score well in similarity metrics. By introducing randomness in thought generation, the model may "get distracted" and fall back to more simple, generic responses. These responses would cluster near the average score, thereby lowering the worst-case performance but potentially boosting the maximum due to the occasional strong match.

Given these findings, for this strategy to have any use, it is crucial to be able to

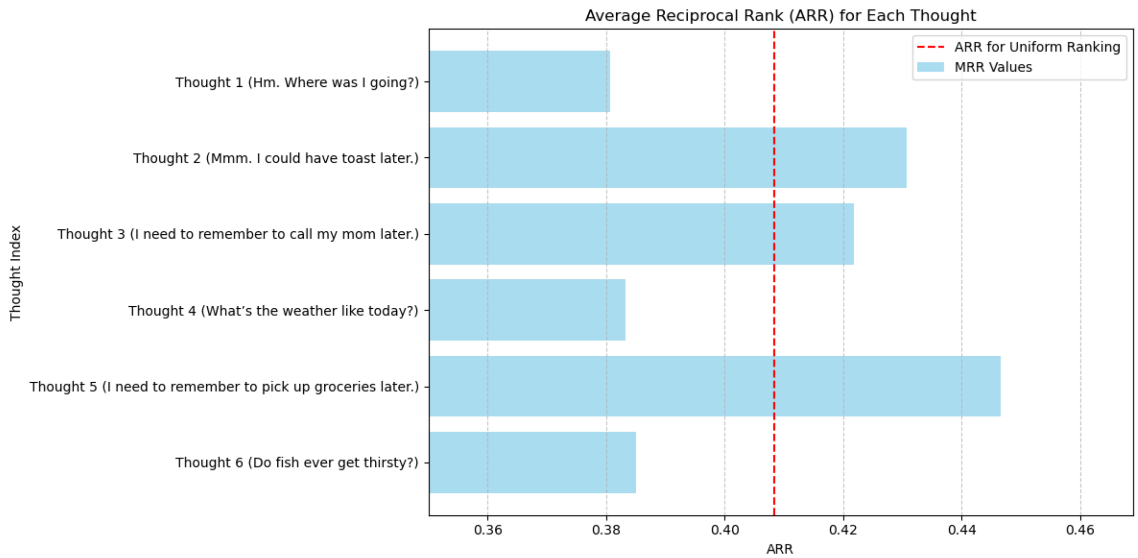

Figure 4.7: Bar chart showing the Average Reciprocal Rank for each random idea subtext. The possible range is between 0 and 1, with a perfect score of 1 being achieved by having a rank of 1 over all samples

interpret performance changes in relation to underlying phenomena, rather than the simple effect of sampling multiple times from a noisy distribution. In this context, further analysis is needed to determine whether improvements in generation quality can be meaningfully linked to real-world mental states.

4.3.5 Using random ideas in subtext does not have clear trends in scores

Given our perturbation analysis, we can examine the random subtext and theory of mind subtext conditions in more depth for interpretability.

Looking at Figure 4.7 and comparing it to 4.5, we can see that the random subtext condition has no clear trend in performance based on the random subtext ideas. There is also more deviation from the uniform ranking ARR value.

Additionally, when computing the average scores across all samples in Figure 4.8, the results are comparable to those for the theory of mind subtexts shown in

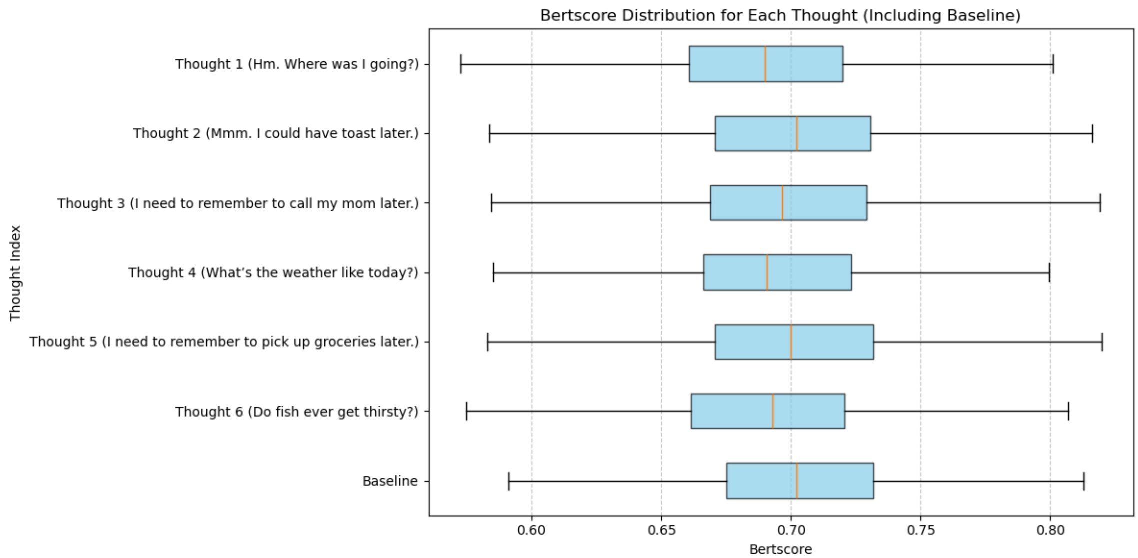

Figure 4.8: Box plot showing the BERTScore distribution for each different subtext over all samples

Figure 4.4. One likely reason for this is that the large amount of conversational context—specifically, the inclusion of 10 preceding turns—allowed the model to produce a sensible reply, even when the subtext and resulting thought were largely ignored. When inspecting the response often they were mainly related to the context and not directly to the subtext thought.

4.4 State Evaluator

This section will review all the selection mechanisms implemented. In summary, no selection mechanism was able to select subtexts based on the conversational context to improve the generation BERTScore.

4.4.1 Choosing the best ranked thought improves BERTScore

To be able to contextualize our LLM-based voting mechanisms, the results from the random and perfect state evaluators are included as a reference for comparison here. A scatter plot of the scores from both state evaluators against the baseline

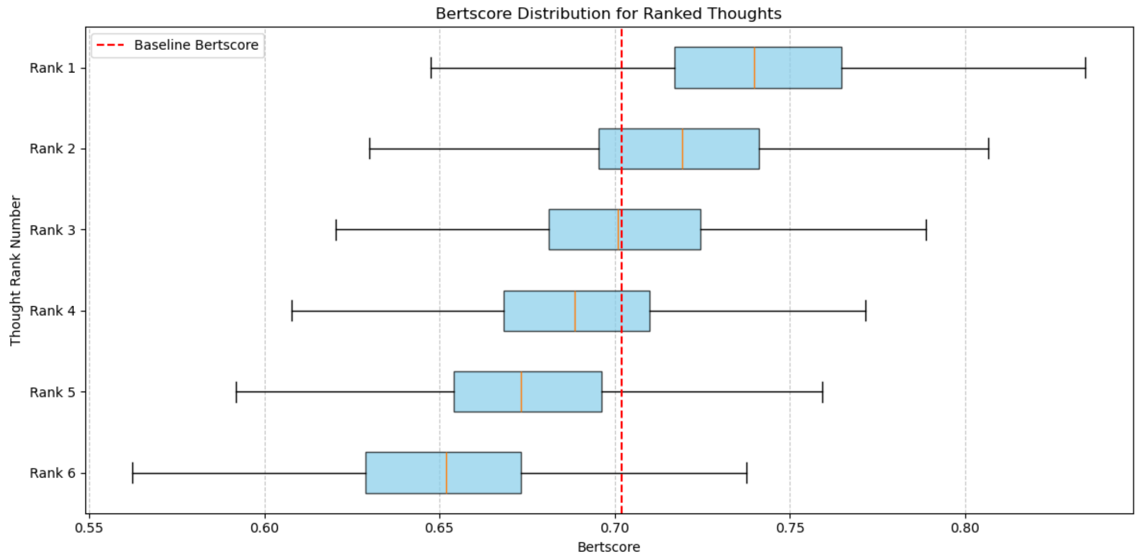

Figure 4.9: Box plot showing the BERTScores for all ranked random idea subtexts (1st, 2nd, 3rd best, etc.) for each sample.

generation score is included in Figure 4.10. We can see that generating a model response based on the perfect state evaluator which chooses the subtext resulting in the best outcome has a BERTScore which is almost always better than baseline single-step generation. However, when selecting a generation based on a random subtext there is no improvement in performance. This is consistent with the data in Figure 4.3 which shows that either the first or the second ranked subtext must be used on average to improve on the baseline.

4.4.2 Both LLM-based state evaluators are not effective in selecting the best subtext

Figure 4.11 depicts the scatter plot of the scores from both LLM-based state evaluators against the baseline generation score. Visually, it seems that the voting mechanism has no noticeable change on the distribution of scores, with the distribution appearing much more similar to the random state evaluator compared to the perfect state evaluator in Figure 4.10.

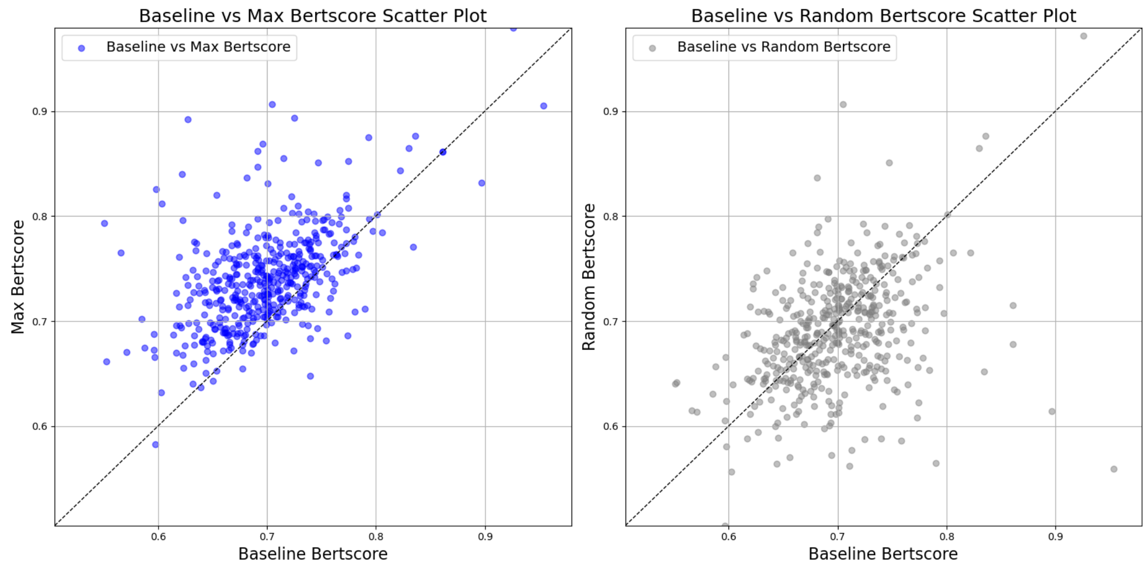

Figure 4.10: Scatter plots of the BERTScore based on model response from the best ranked thought (left) and a randomly selected thought (right) against baseline response

The results in Figure 4.12 confirm these observations. All selection mechanisms apart from the perfect state evaluator perform marginally worse than the baseline. It can be seen that the perfect state evaluator also has a slightly less spread distribution.

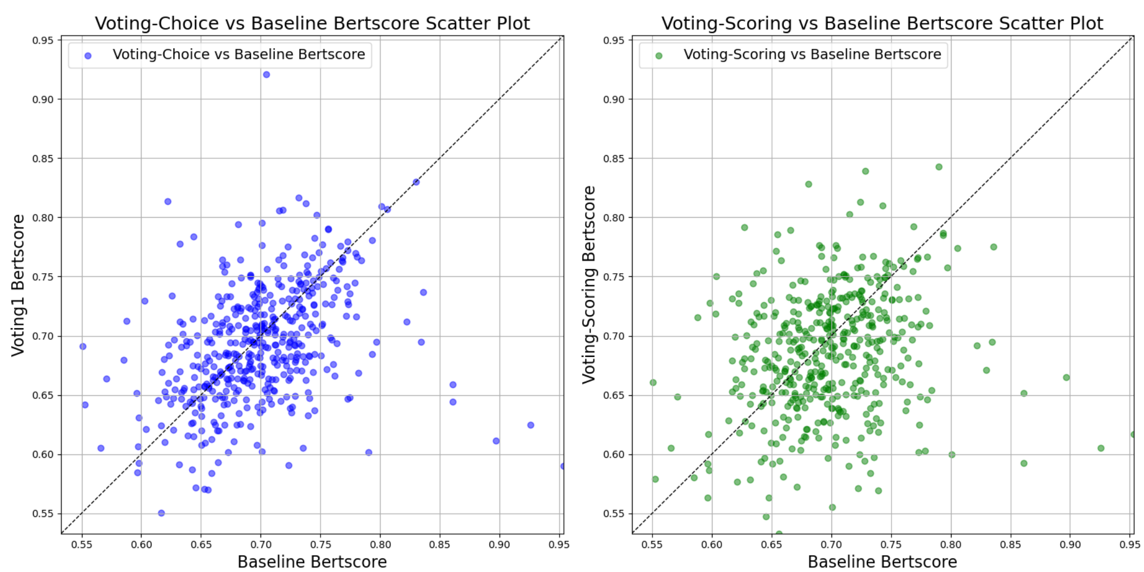

Figure 4.11: Scatter plots of the BERTScore based on model response from the voting over all possible subtexts (left) or scoring each subtext individually (right) against the baseline response.

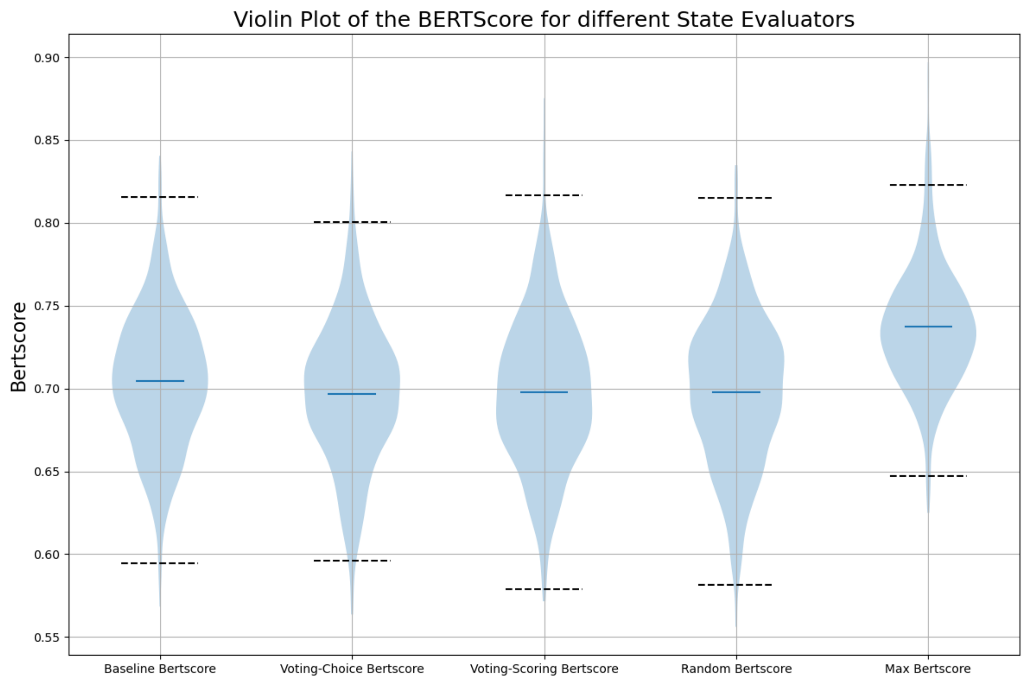

Figure 4.12: Violin Plot of the BERTScore for different State Evaluators. The dark blue line represents the mean of the distribution while the black lines represent the datapoints 1.5 times the interquartile range (IQR) from the first and third quartile.

4.5 Evaluation over multiple shuffles

The following results are obtained by applying the voting scoring method and voting choice method 10 times each with the order of subtexts to choose randomized each time.

4.5.1 Evaluation by Scoring and by Voting are biased towards certain numbers

One reason that the state evaluation mechanism is not working well could be because of bias, which can be an issue when using LLMs as evaluators [74][75]. An obvious way this appears is a preference for certain numbers.

In Figure 4.13 we can see that for example the LLM perfers not to choose the first item in the list regardless of which subtext it represents. Similarly, in Figure 4.14 we can see that the scoring mechanism has a strong preference for the digits 2,4 and 8. With a couple other digits having frequencies one or multiple orders of magnitude less, since the graph is plotted using a logarithmic scale.

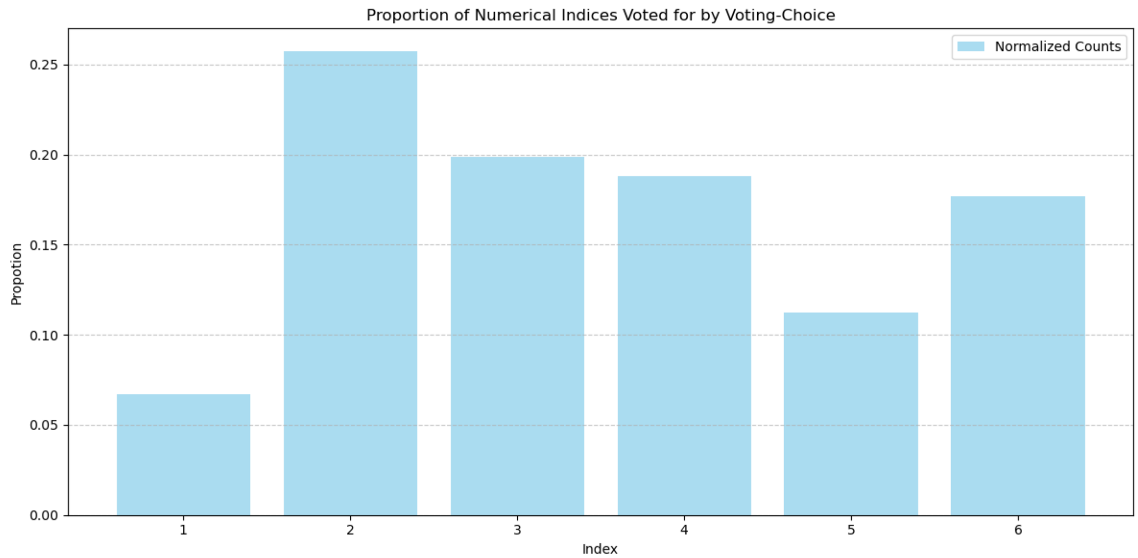

Figure 4.13: The frequency each numerical index in the list is chosen when the state evaluator votes over states

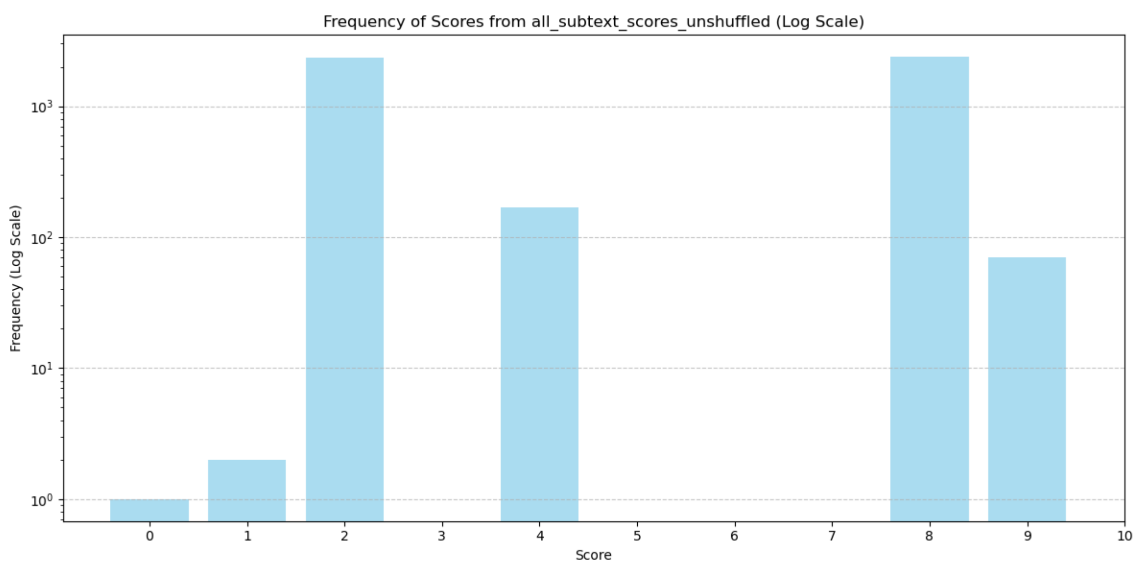

Figure 4.14: The frequency specific digits appear when the state evaluator scores each option. Calculated over 10 shuffles.

4.5.2 Evaluation by Scoring is more consistent than by Choice

We can also quantify the consistency of the voting mechanism by using the metrics defined in Section 3.3, evenness index, and maximum agreement score.

Figures 4.15 and 4.16 show the dstribution of these metrics over all the samples. It is clear that both voting mechanisms have high levels of bias; for example, for most samples, the maximum agreement on the best subtext over the different shuffles did not each a majority of more than 4. However, these plots do show that the evaluation method by scoring independently is more consistent than voting over all the possible options, since the average entropy is closer to 0 and the number of maximum agreements also increases.

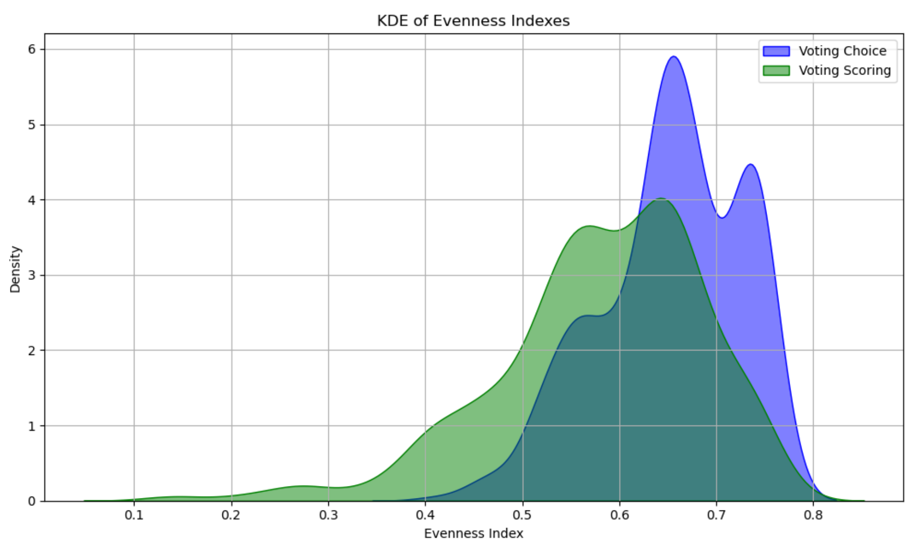

Figure 4.15: The continuous density plot of evenness index of all samples for different shuffles. An evenness index of 0 would indicate perfect consistency, while an index of 1 would indicate a perfectly random/uniform evaluation process.

4.6 Majority Voting

Since we implemented a shuffling mechanism, we can also create a meta-state evaluator by selecting the most popular subtext after a certain amount of shuffles.

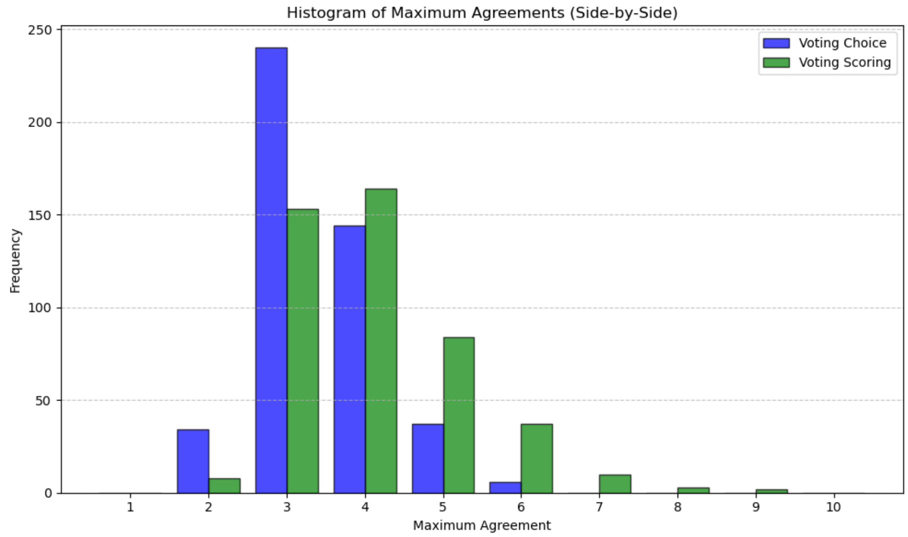

Figure 4.16: The histogram of maximum number of agreements on the best subtext over different shuffles for all samples. A score of 10 would indicate perfect consistency while a uniform/random evaluator would score 1.67 on average.

4.6.1 Majority voting does not improve performance

In the scatter plots in Figures 4.17 and 4.18 we can see that sampling the state evaluator multiple times does not have a noticeable improvement on the distribution of scores.

4.6.2 High consistency on evenness index leads to lower variation of BERTScore of output

Figure 4.19 shows the distribution of scores for the majority vote meta-state evaluators as well as the distributions of the samples evaluated with high consistency (which are also highlighted in the scatter plots in Figures 4.18 and 4.17). While neither the majority vote meta-State Evaluators or the high consistency examples have a greater similarity to the human reference compared to the baseline, the evenness index metric of the scores or voting is indicative of the variance of the final BERTScore distribution. That is, the distribution of high consistency examples plotted as judged by evenness

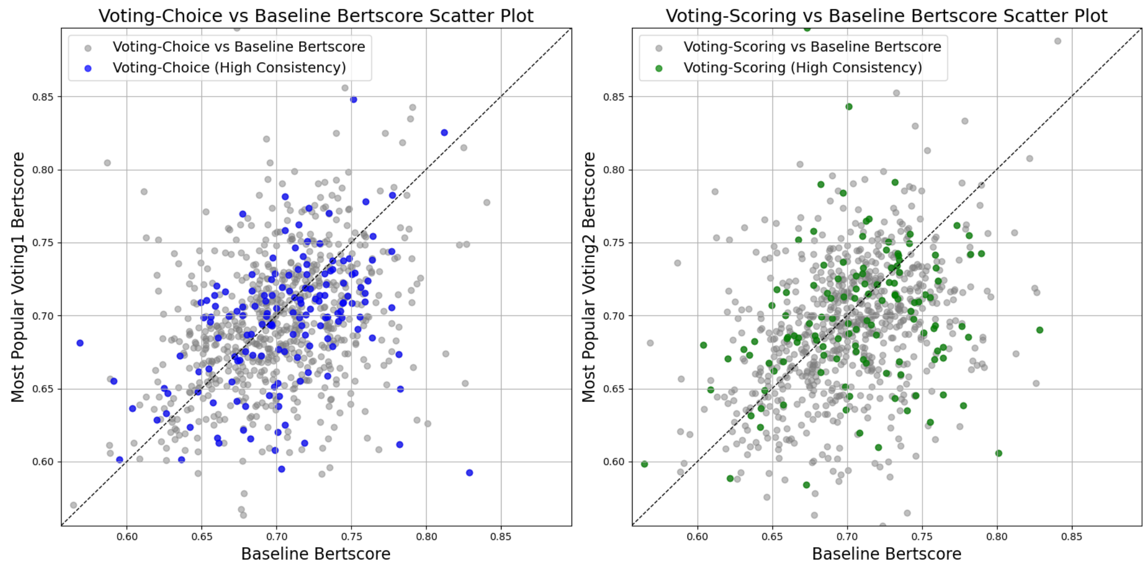

Figure 4.17: A scatter plot of the BERTScore of the majority vote meta-state evaluator voting by voting and by scoring. General points are in grey while coloured points refer to high consistency examples with number of agreements ¿3.

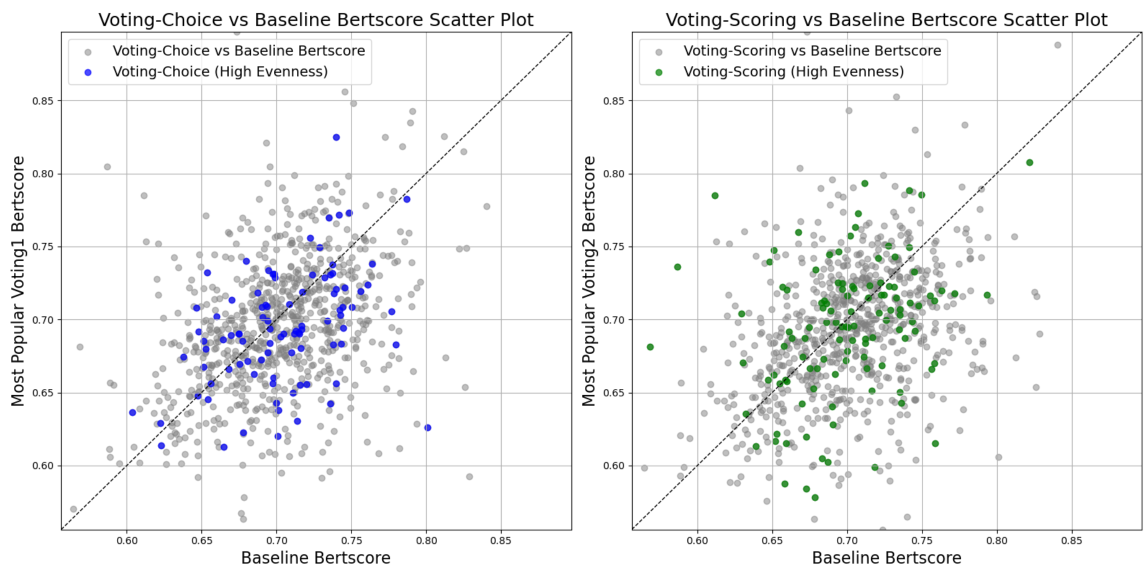

Figure 4.18: A scatter plot of the BERTScore of the majority vote meta-state evaluator voting by voting and by scoring. General points are in grey while coloured points refer to high consistency examples with evenness index0.55.

index has a lower variation of BERTScore compared to all other conditions.

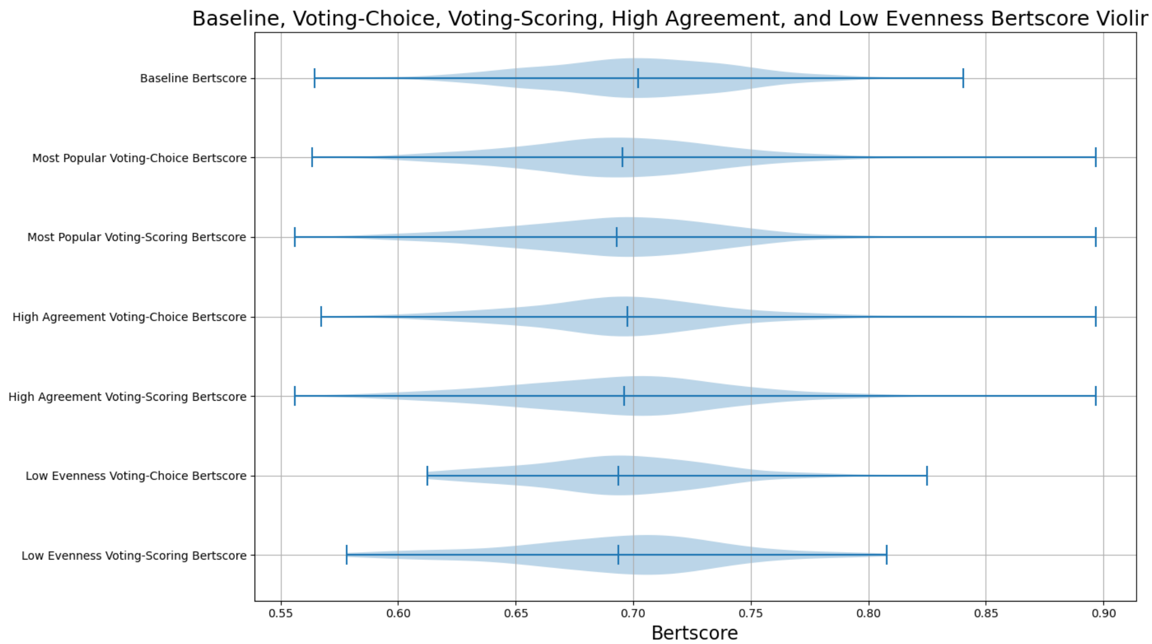

Figure 4.19: Violin Plot of the BERTScore for the majority vote meta-State Evaluators based on scoring and voting. Also plotted are the distributions of high consistency examples highlighted in the scatter plots in Figures 4.18 and 4.17

4.7 Overall findings

In this project, we further demonstrated that large language models are capable of generating sensible and coherent dialogue. Our experiments showed that both model size and context length positively correlate with similarity to human-generated references.

We introduced a thought generator based on a theory-of-mind taxonomy, which produced diverse response sets—some outperforming and others underperforming the baseline. This motivated the implementation of a selector mechanism designed to choose the most promising subtext to guide response generation. To better understand the generator's behavior, we conducted a perturbation analysis, exploring how changes to different components influenced the distribution of model outputs. While BERTScore appeared to be a reasonable metric for assessing textual similarity, further investigation is needed to determine whether it captures sufficient semantic nuance, and whether the generated thoughts and responses genuinely align with the theory-of-mind taxonomy provided.

Finally, we developed three state evaluation methods to select the best subtext for response generation. However, none of these approaches outperformed the baseline model that generates responses in a single step. Interestingly, we observed inconsistency and bias in the voting and scoring evaluators, though the consistency of an evaluator across multiple data shuffles—measured using the evenness index—appeared to reflect the spread in BERTScores of the model outputs. This suggests that LLMs might have some capacity to internally assess or express confidence in their generated responses.

Chapter 5

Discussion

While the ultimate goal of automatically improving dialogue generation using a theory of mind-based in-context learning approach was not fully achieved, this project still provides valuable insights—both for future improvements and for our broader understanding of conversational modeling.

To better understand the performance and shortcomings of the system, we can break down the implementation into its key components. Given the complexity and number of moving parts, this component-wise reflection is essential for identifying areas for refinement.

5.1 Future work

5.1.1 Language Model Capabilities

First, the capabilities of the language model itself, denoted p_θ, may have been a limiting factor. It is possible that the model's social reasoning abilities are insufficient for accurate inference in this domain. This challenge is compounded by the inherent noisiness and ambiguity of social reasoning tasks. If the LLM has not been trained extensively on such data—likely, given the difficulty of curating high-quality theory-of-mind datasets—it may lack the inductive biases required for these tasks. Although the weights of the LLaMA 3.x models are publicly available, their training data is not, making it difficult to verify this hypothesis. Potential next steps include fine-tuning the model on curated social reasoning datasets, such as SocialIQA [76], or designing a new benchmark of simpler, well-scaffolded theory-of-mind tasks to evaluate and adapt the model progressively.

5.1.2 Context Representation

Second, the context c provided to the model may not have been rich or structured enough. While it included recent conversational history, many other types of information are relevant for generating socially coherent dialogue: background world knowledge, participant personalities, shared memories, and conversational goals. A more structured or explicitly abstracted representation of context may better support theory-of-mind reasoning, aligning with thoughts in other work [26].

5.1.3 Evaluation Metrics.

Third, the evaluation metric—BERTScore—may lack the granularity necessary to capture nuanced social reasoning or alignment with human intent. Although BERTScore correlates with textual similarity, it does not assess propositional content and does not have extremely high correlations with human judgment in other tasks. Alternative metrics could include: using a pipeline to convert responses and labels to propositional form, to remove any reliance on surface represetnation of text before using a text similarity like BERTScore. Additionally, encoding accuracy from the sentence embedding of the generated text could be used as a score of how good a generation is. Then BERTScore as a metric could also be validated by seeing how well it correlated with encoding accuracy. Perplexity could also be used as a metric to capture likelihood of a response given context. Finally, human evaluations canbe used to validate the use of these automatic scores.

5.1.4 Thought Representation and Tree of Thought.

Another area for exploration is the structure of the thought process itself. It remains unclear whether the intermediate thoughts generated by the LLM accurately reflect the model's internal reasoning [77, 78, 79, 80]. Doing more perturbations of thoughts could help show this.

An especially compelling dataset to apply this method to is ESConv [81], which features emotional support conversations between a "speaker" describing their problems and a crowdworker "listener" providing support. Each response from the listener is annotated with a communicative strategy, selected from a set that includes Questions, Self-disclosure, Affirmation and Reassurance, Providing Suggestions, Other, Reflection of Feelings, Information, and Restatement or Paraphrasing. This makes ESConv a rich resource for evaluating whether social reasoning capabilities—such as those enabled by a tree-of-thought approach—can be leveraged to predict the underlying social strategies used by the listener.

Furthermore, one potential direction is to simplify the thought process, for instance by reducing the tree depth from two to one, and conditioning the final response more directly on the intermediate reasoning steps. Prompt engineering—particularly for generating or interpreting theory-of-mind subtext—could also be refined.

5.1.5 Filtering to Target Turns that Require Reasoning

An important open question in conversational analysis is determining when participants engage in more automatic, reflexive responses versus when they employ deliberate reasoning. Our current predictions may capture turns that elicit automatic replies, which are inherently more challenging to model due to their low cognitive load and variability. Incorporating more sophisticated filtering methods could help isolate conversational turns that genuinely require thought or reasoning. For instance, the high BERTScore of the generic label "Yeah I see that too" in Section 4.3.4 suggests that such low-effort responses may be disproportionately influencing our evaluation metrics, indicating a need for refined filtering criteria.

5.2 Limitations

First, the project assumes that conversations can be segmented into rigid, alternating turns. While this simplifies implementation, it does not reflect the fluid nature of real-world dialogue, which often includes interruptions, overlaps, or simultaneous speech. Moreover, it overlooks how interlocutors build shared context over time. The common ground theory [82] describes how shared knowledge accumulates and simplifies communication—for example, initial explanations may be detailed, but future references become more abbreviated. While this project began with segmenting dialogues into discrete two-person turns, future work could explore continuous, overlapping, or multi-party dialogue representations.

Second, the model is fundamentally unimodal—it operates on language alone. However, in human conversation, responses are influenced not only by the preceding words but also by a range of visual, situational, and embodied cues. Listeners naturally integrate information such as visual scene understanding [83], the goals and perspectives of the speaker [84], the physical context or affordances of objects in the environment [85], and paralinguistic signals such as facial expressions or body language [86, 87]. While some of these aspects may be indirectly encoded in language, much of this contextual richness remains implicit and inaccessible to language-only models. As such, future modeling efforts could benefit significantly from incorporating multimodal inputs or grounding the model in real-world context.

Chapter 6

Related Explorations-Neural Encoding using Sentence and Word Embeddings

This chapter presents the second project of the thesis: exploring how to perform sentence-level encoding of intracranial neural recordings. Unlike fMRI, intracranial recordings offer much higher temporal resolution—often far exceeding the duration of a single sentence. This mismatch in timescales makes sentence-level alignment particularly challenging.

Previous work on this dataset has primarily focused on encoding neural activity using token-level embeddings. However, incorporating sentence embeddings could provide a window into higher-order cognitive processes by capturing more abstract semantic representations. In this section, we investigate the feasibility of combining word and sentence embeddings to model neural activity using a banded ridge regression framework.

6.1 Data collection and pre-processing

The original dataset and pre-processing method is provided by Zada et al. (2025) [88]. To summarise, a 30 minute audio story (podcast) was played to nine participants while undergoing electrocorticographic monitoring for epilepsy. A simple preprocessing pipeline was implemented to extract the high-gamma band power per electrode. Alongside the neural recording, the story was also manually transcribed with each word timestamped at high temporal resolution.

The stimulus presented to participants was a segment from the podcast "This American Life" entitled "So a Monkey and a Horse Walk Into a Bar: Act One, Monkey in the Middle" released on November 10, 2017. The original audio and transcript are freely available online (https://www.thisamericanlife.org/631/transcript). We manually transcribed the story and timestamped words at high temporal resolution.

Raw electrode data underwent the following preprocessing pipeline, which is quoted from the Zada et al [88]. The steps include "removing bad electrodes, downsampling, despiking and interpolating high amplitude spikes, common average re-referencing, and notch filtering. First, we visualized the power spectrum density of each electrode per subject. From this, we were able to annotate unusual electrodes that did not conform to the expected 1/f pattern, had a consistent oscillatory pattern, or showed other unusual artifacts. We found 31 such electrodes and marked them as "bad" (identified in the accompanying metadata). The source of these artifacts may be due to several factors, including excessive noise, epileptic activity, no noise, or poor contact. For data acquired with a sampling rate greater than 512 Hz, we downsampled to 512 Hz to match the sampling rate across subjects. We then applied a despiking and interpolation procedure to remove time points that exceeded four quartiles above the median of the signal and refill it using pchip interpolation. For re-referencing, we subtracted the mean signal across all electrodes per subject from each of their individual electrode time series. Finally, we used notch filters at 60, 120, 180, and 240 Hz to remove power line noise. This pipeline produces a "cleaned" version of the raw signal."

6.2 Method: Ridge regression

Here, I will describe the original method in the paper for ridge regression to decode neural activity from token-level embeddings.

Let $X \in \mathbb{R}^{n \times p}$ be a feature matrix with n samples and p features, $y \in \mathbb{R}^n$ a target vector, and $\alpha > 0$ a fixed regularisation hyperparameter. Ridge regression defines the weight vector $b^* \in \mathbb{R}^p$ as:

$$b^* = arg \min_b ||Xb - y||_2^2 + \alpha ||b||_2^2. \tag{6.1}$$

The simplest way of casting this problem would be to have n be the number of words and p be the number of electrodes, so that the target matrix Y has length of number of electrodes. Each entry in X would contain the word embedding for the word at each new word onset, while Y would contain the voltage signals recorded from the electrodes placed on the brain's surface in Volts. ECoG electrodes measure the aggregate electrical activity of thousands to millions of neurons and the recorded will mostly reflect local field potentials (LFPs), which arise from the synchronous activity of many neurons. Additionally, if the word consists of more than one token, we average all of them to get the embedding per word.

However, this method is unideal because the electrode data (512Hz) is much higher frequency than the spoken words (2Hz). To solve this, we epoch the electrode data around the onset of each word, so that X now has shape (number of words, number of ECoG electrodes x number of lags) and Y has shape (number of ECoG electrodes x number of lags). The epoch period chosen is from -2 seconds to +2 seconds relative to word onset. This is equivalent to fitting a separate regression model for each electrode and each epoch.

Before the model can be fit, there are there are two additional elements in the pipeline we define in Scikit-learn [89]. First, Y and the regressors in X were mean-centered using StandardScaler. Second, because the design matrix was wider than it is long, we used the kernel method to solve the ridge regression in its dual form [90]. Specifically, we used a linear kernel for each feature space separately before fitting the model.

The penalty term b is learned using an inner cross-validation setup within the training set using the himalaya python library [91]. Then an outer cross-validation loop is used to evaluate the encoding model with $k = 2$. We evaluated the performance of each encoding model on the held-out fold by correlating the model-predicted ECoG signal Y_{preds} with the actual ECoG signal Y_{test} in the test fold. Finally, we averaged the two correlations for each fold to obtain one correlation value denoting the encoding performance for each feature space, electrode, and lag.

6.3 Method: Banded Ridge regression

We use a banded ridge regression model to regress to get the ECoG signal in Y with two different feature spaces with their own regularisation parameters as follows.

$$b^* = \arg\min_b \left\| \sum_{i=1}^{m} X_i b_i - y \right\|_2^2 + \sum_{i=1}^{m} \alpha_i \|b_i\|_2^2 \tag{6.2}$$

The variables are similar to Equation 6.1, where X_i represents the feature matrix corresponding to the i-th group, b_i denotes the coefficient vector for the i-th group, α_i is the regularization parameter specific to the i-th group and$m$ is the total number of feature groups.

In practice for constructing X, the shape of X_1 and X_2 are both (number of words, number of ECoG electrodes x number of lags). Because a sentence contains multiple words, the sentence embedding will be the same for each word in that sentence.

The embeddings used are:

- **Static word embeddings:**
  - Word2Vec embeddings (300-dimensional)
- **Contextual word embeddings:**
  - GPT-2 (13 layers of embeddings, each word embedding is 768-dimensional)
- **Sentence embeddings:**
  - SentenceTransformer `intfloat/e5-mistral-7b-instruct` (4096-dimensional)

6.4 Results

6.4.1 Encoding with context and sentence embeddings outperforms static embeddings, but not contextual embeddings

Figure 6.1 depicts the encoding performance of a banded ridge regression with joint contextual and static embeddings compared to a ridge regression with either only static or contextual embeddings. It is clear that the joint embedding model improves on the static embeddings however compared to the contextual embeddings the performance appears very similar.

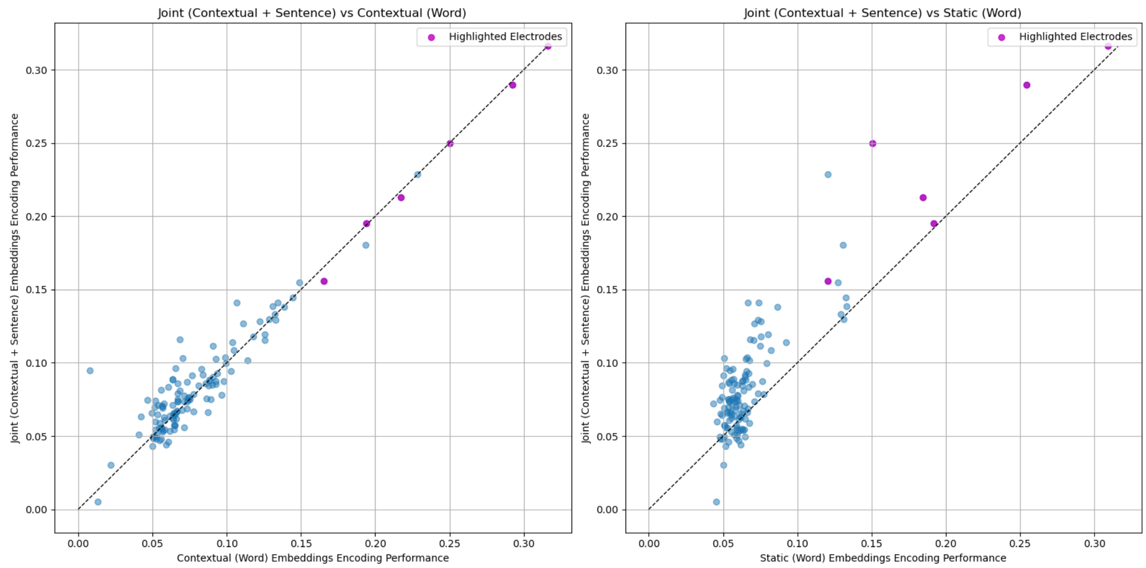

Figure 6.1: Scatter plots of encoding performance (correlation) for joint (sentence+contextual word) embeddings against static and contextual embeddings

6.4.2 Sentence Embeddings take up variance in Joint Context+Sentence encoding models

While using banded ridge regression with contextual and sentence embeddings does not improve upon the performance of ridge regression using only contextual embeddings, examining how the variance is split in the joint model can still be ueful.

Figures 6.5 and 6.6 and show that variance is split to a large extent between contextual and sentence embeddings. Figure 6.5 shows the encoding performance with either feature space separately, which shows that by themselves the overall encoding performance decreases. However, there are some electrodes where the sentence embeddings have a higher relative encoding performance and others where contextual embeddings have a much higher encoding performance on the right of the graph.

Plotting the encoding performance on the brain in Figure 6.6 we can see that the electrodes where the contextual embeddings do best are around XYZ brain areas. However, the correlation for sentence embeddings throughout the brain is generally more diffuse.

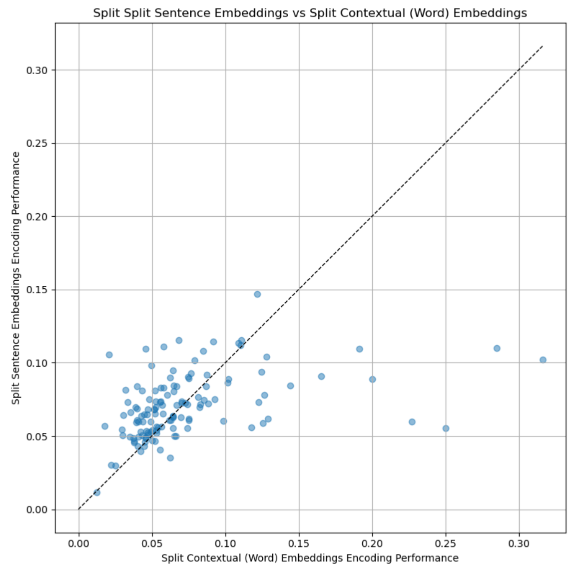

Figure 6.2: The encoding performance using either the contextual embedding or sentence embedding feature space plotted against each other

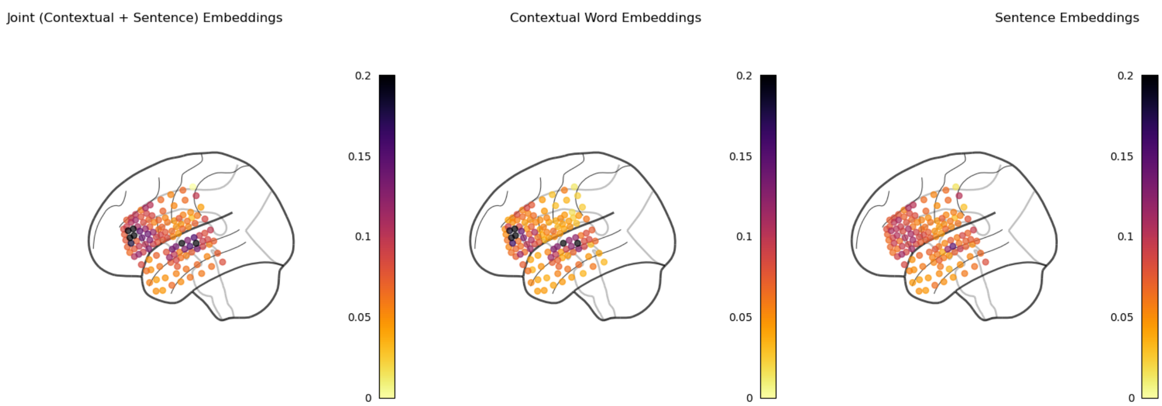

Figure 6.3: Correlation scores per electrode mapped onto the brain for joint (contextual word + sentence), contextual word and sentence embeddings

6.4.3 Encoding with static and sentence embeddings outperforms static embeddings, but not contextual embeddings"

In contrast to Figure 6.1 we compare the encoding performance of joint (static + sentence) embeddings compared to contextual and static embeddings in Figure 6.4

as opposed to joint (contextual+sentence) embeddings.

In Figure 6.4 we can see that the joint (static+sentence) model performs worse than model based on just contextual embeddings, however this joint model improves performance on models trained with just static embeddings. Most of this increase in performance happens in a range between a correlation of 0.05 to 0.09 for the static model, which increases to a range of 0.05 to 0.13. The performance for high correlation electrodes tends to be similar for both static and sentence embeddings.

6.4.

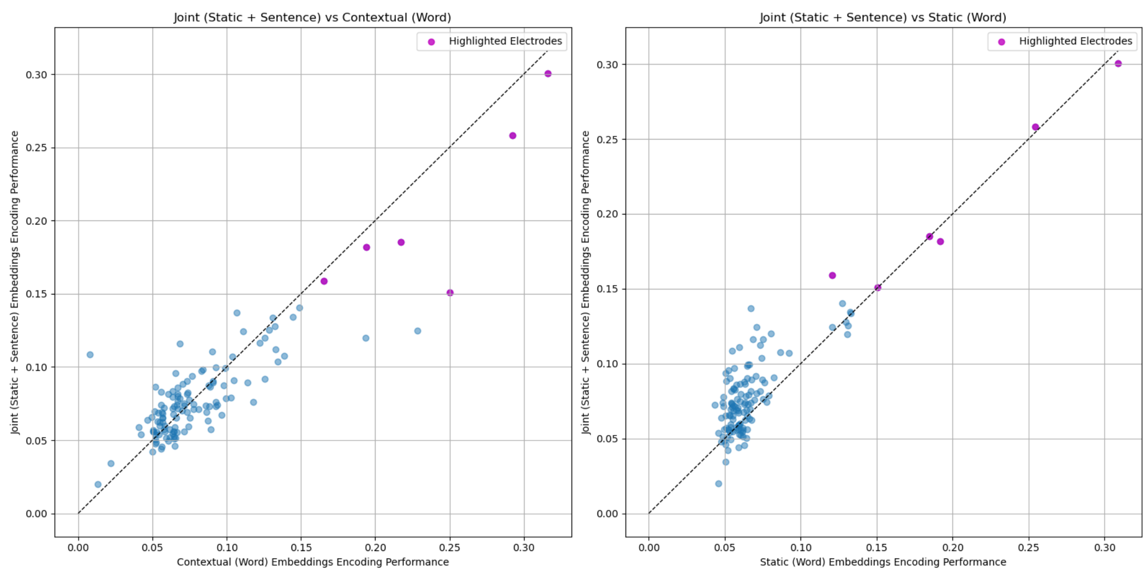

Figure 6.4: Scatter plots of encoding performance (correlation) for joint (sentence+static word) embeddings against static and contextual embeddings

6.4.4 Joint Sentence+static embeddings enable encoding for best electrodes over a greater period of time compared to only static embeddings

Another way of plotting our results is by collapsing the electrode dimension by selecting a subset and averaging, rather than collapsing the time dimension by choosing the maximum correlation over all time.

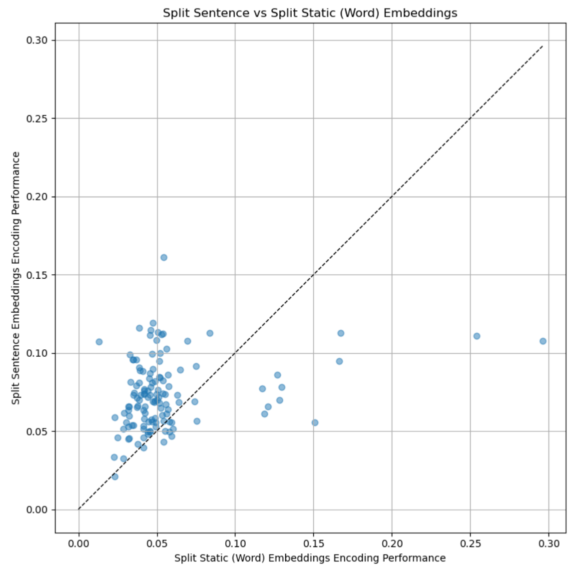

Figure 6.5: The encoding performance using either the contextual embedding or sentence embedding feature space plotted against each other

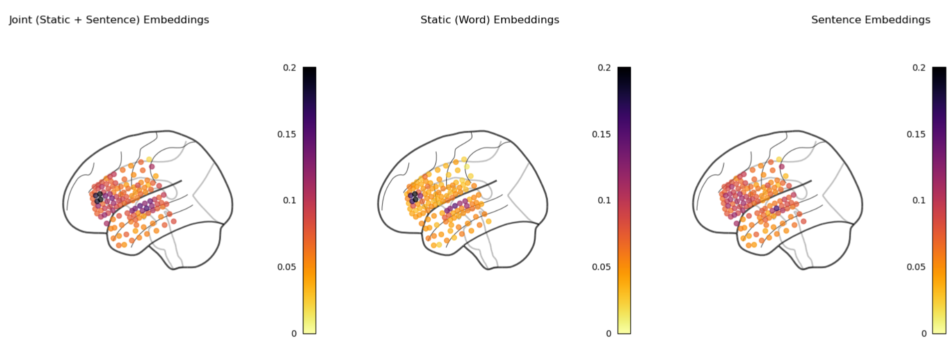

Figure 6.6: Correlation scores per electrode mapped onto the brain for joint (static word + sentence), static word and sentence embeddings

Figures 6.7 and 6.8 show the correlation over time for a subset of the best-performing electrodes. From Figure 6.7 is clear that encoding using sentence embedding in addition to static embeddings increases the encoding performance over a greater period of time. In Figure 6.8 we can see this effect as well, but it is less noticeable because the contextual embeddings already perform better over a greater period of time compared to the static embeddings. Nevertheless, by looking at the split embedding spaces we can see that the word embeddings generally peak a few

100ms after word onset, but the encoding accuracy using sentence embeddings is more spread out over time.

To be able to relate the size of the outer blue envelopes in Figures 6.7 and 6.8 we also plot them overlapping alongside the results from contextual embeddings in Figure 6.9(a). We can see that the outer envelopes of contextual, contextual+sentence and static+sentence have similar profiles. While in Figure 6.9(b) we can see that encoding using either static or split-static embeddings has a more narrow profile.

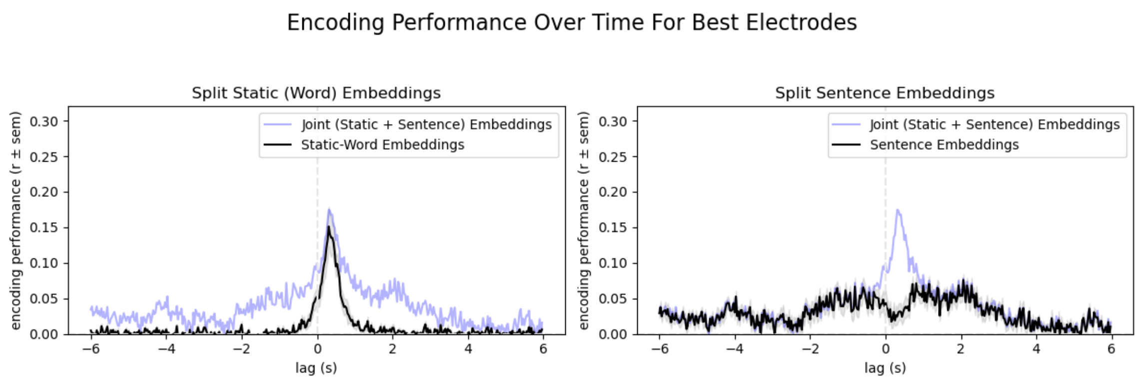

Figure 6.7: The correlation of model and actual neural signal plotted over time and averaged over best performing electrodes. Black lines show the encoding done using a split feature space after banded ridge regression for static embeddings (left) and sentence embeddings (right). The blue line represents the joint embedding performance. Electrodes where the joint (static+sentence) correlation had a performance of more than 0.15 were selected.

6.4.5 Overall findings

In this project, we successfully implemented a model for sentence encoding. We found that static and sentence embeddings when combined can get encoding performance that approaches that of contextual embeddings, with the added flexibility of being able to change the content of the sentence embedding easily compared to contextual word embeddings.

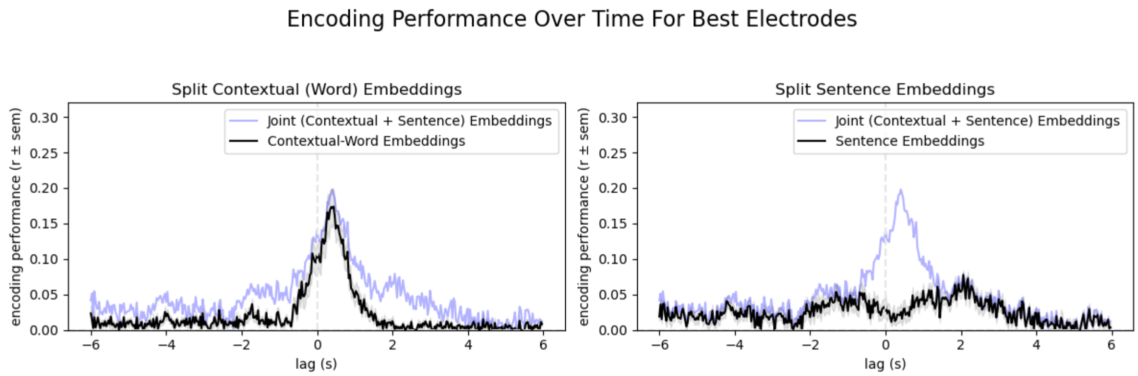

Figure 6.8: The correlation of model and actual neural signal plotted over time and averaged over best performing electrodes. Black lines show the encoding done using a split feature space after banded ridge regression for contextual embeddings (left) and sentence embeddings (right). The blue line represents the joint embedding performance. Electrodes where the joint (static+sentence) correlation had a performance of more than 0.15 were selected

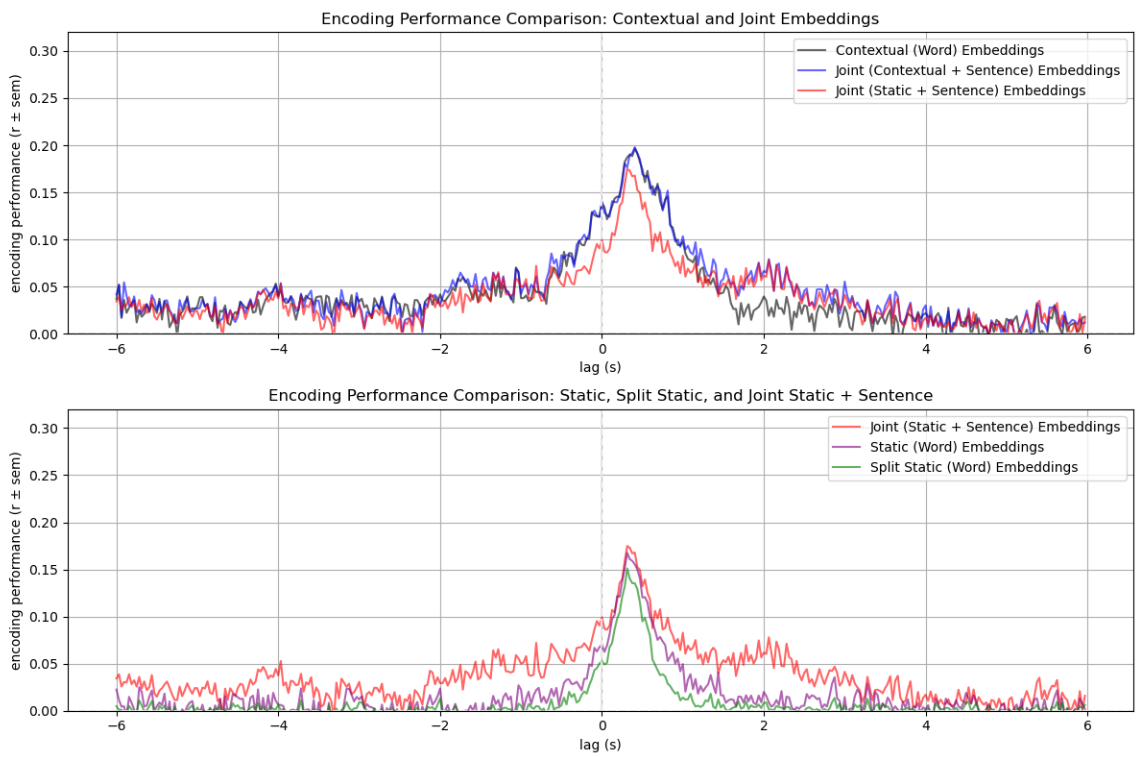

Figure 6.9: The correlation of model and actual neural signal plotted over time and averaged over best performing electrodes. Embeddings used in the model in order of envelop size are contextual, joint (contextual+sentence), joint (static+sentence) (plotted twice), static, and split static.

Chapter 7

Future Directions

Now that the two projects in this thesis have been established, it is worth exploring how they might be integrated to offer insight into the neural basis of theory of mind.

One direct way to bridge the conversational modeling project with neural data is by using the voted subtexts as labels for each utterance. These labels could serve as supervision signals in a neural classifier aimed at predicting the mental state most likely associated with each utterance. A crucial aspect of this task would involve distinguishing between moments when a person is engaged in mental state reasoning versus more routine conversational behavior—an open and intriguing question.

If the dialogue generation task proves challenging, an interesting alternative would be to adapt ideas from masked sentence modeling. Rather than predicting the final utterance in a conversation, the model could instead predict a missing utterance from the middle. This would allow the model to leverage both past and future context, potentially resulting better ability to reason over theory of mind.

In terms of evaluation, rather than relying solely on BERTScore, one could assess how strongly generated utterances drive activity in neural encoding models. After training on a subset of ground-truth data, such an approach could help identify "supernormal stimuli"—stimuli that elicit exaggerated responses from a given neural system, as described by Barrett (2010) [92].

This project also lends itself naturally to an in-context learning framework, and its application need not be limited to social reasoning alone. Insights from psycholinguistics could enrich the modeling, particularly theories related to how people construct situation models through bridging inferences during discourse. Some accounts propose that literal and non-literal meanings are computed simultaneously within a parallel distributed framework—phenomena that could be explored by prompting large language models and examining their alignment with brain activity.

Another compelling area for exploration is inner speech. Inner speech serves diverse functions such as deliberation, planning, clarification, problem-solving, and even self-regulation or self-motivation (e.g., "You can do this," "Don't do that") [93]. Modeling these internal dialogues might provide a novel pathway for connecting language-based AI models with neural data, offering a richer picture of cognition and self-directed thought.

Chapter 8

Conclusion

This thesis explored the extent to which Large Language Models can perform socially informed reasoning during dialogue, focusing on the inference of subtextual thoughts—unspoken beliefs, desires, and intentions—through a theory-of-mind-inspired approach. By applying the Tree of Thought framework to naturalistic dialogue generation, we demonstrated that LLMs are capable of generating plausible, socially coherent responses. However, our results also revealed that existing evaluation metrics and selection mechanisms struggle to consistently identify the most appropriate subtext or response, highlighting limitations in both model expressivity and current assessment tools.

In the second part of this thesis, we extended LLM applications to neural encoding, showing how both word- and sentence-level embeddings can be used to model brain activity during language comprehension. This builds on prior work in the Hasson lab and offers new directions for integrating computational representations of language with neuroscience.

Across both projects, we found that while LLMs exhibit emerging capacities for modeling literal and inferred content, their reasoning processes remain opaque and prone to inconsistency. Evaluator bias, limited context representation, and inadequate scoring methods present ongoing challenges. Nonetheless, the promising signals observed in perturbation and consistency analyses suggest that LLMs may possess latent capacities for internal assessment and social inference that are yet to be fully unlocked.

Future progress will likely depend on more structured representations of context, improved datasets for training and evaluation, and new methods for interpreting model reasoning. Ultimately, this work contributes to the growing effort to bridge the gap between surface-level language modeling and deeper social cognition in AI systems, with implications for both computational neuroscience and the development of more human-aligned dialogue agents.

Appendix A

Code Implementation

A.1 System prompt used

You will be given an excerpt of a conversation that includes the next speaker. It will be formatted as Speaker 1: message, Speaker 2: message. Reply as the next speaker in the conversation only. Format your reply as Speaker 1: message. Do not say you are a chatbot.

A.2 Prompt Generation

```
{

import json
import random

#final JSON file format:
#prompts_subtext: a list of length s (number of subtexts) containing
    lists of length n (number of steps)
#subtexts: a list of length s containing the subtexts
```

```python
#prompts_subtext_voting: a list of length 3 containing the prompts for the subtext generation (list of length s), voting (single string), and next speaker sentence generation (single string)



#numer of subtexts=s, number of steps=n

def exportJSONFile(data,fileName):
    json_object = json.dumps(data)

    with open(fileName, "w") as outfile:
        outfile.write(json_object)


subtextPrompt="You will be given an excerpt of a conversation between two speakers. Format your output as {Output:}. Your task is to generate the subtext thought that the next speaker is having, based on the following idea:"

#length s
subtextIdeaList=["Do I agree with this?","What do they want from me?","How are they feeling right now?","What are they avoiding mentioning?","W h a t s  their body language saying?","Should I clarify my earlier point?","What do they really mean by this?"]
#subtextIdeaList2=["Why are they telling me this now?", "What triggered this topic in the conversation?","How does this relate to what they said earlier?","Does this connect to something t h e y ve  shared about themselves before?","How did they accomplish this, or how are they planning to?","What resources or skills did they use to solve this problem?","What are the implications of this for me or them?","What are they not saying
```

```
that might be important?","What are they trying to achieve by saying this?","How does this align with their long-term goals or ambitions?"]
#subtextIdeaList3=["What are they about to say or ask?","Are they feeling frustrated, happy, or anxious?","Is this behavior typical for them, or is it out of character?","Are they hinting at a deeper issue without explicitly saying it?","Are they trying to persuade me?","Their background might explain why they see things this way."]
#subtextIdeaList4=["How do I feel about this?", "How does the other person feel about this?", "What do I want to happen next?", "What do they want to happen next?", "What do I need to know to make a decision?", "What do they need to know to make a decision?"]


randomSubtextList=["Hm. Where was I going?","Mmm. I could have toast later.","I need to remember to call my mom later.","Whats the weather like today?","I need to remember to pick up groceries later.","Do fish ever get thirsty?"]


subtextIdeaList_only4=["What do I want to happen next?", "What do they want to happen next?", "What do I need to know to make a decision?", "What do they need to know to make a decision?"]

prompt1=[]
prompt1Random=[]
selectedSubtexts=subtextIdeaList_only4
print(f"The number s of subtexts is {len(selectedSubtexts)}")
for i in range(len(selectedSubtexts)):
    #try this later (include info in the content) prompt1.append({"role": "system", "content": subtextPrompt + " " + s})
```

```
    prompt1.append(subtextPrompt + " " + selectedSubtexts[i])
    prompt1Random.append(subtextPrompt + " " + randomSubtextList[i])


#promptVoting="Based on the subtext generated, select the thought
   that you think is most likely to be the subtext of the next
   speaker. Output a single number corresponding to the thought you
   choose. Output this number as {Output:num}"
promptVoting="Based on the subtext generated and the conversation
   history provided, select the thought that is most likely to be
   the subtext of the next speaker. Output a single number
   corresponding to the thought you choose. Output this number as {
   Output:num}. "

promptVotingScoring="Based on the subtext generated and the
   conversation history provided, output a score corresponding to
   the likelihood that the subtext is the true subtext of the next
   speaker. Output this score as a number from 1 to 10 as {Output:
   num}. "


#or can include a newly formatted prompt that just contains relevant
    info (does not include the last prompt)
#prompt2A="You will be given an excerpt of a conversation that
   includes two speakers. It will be formatted as {Speaker 1:
   message, Speaker 2: message}, followed by {Thought:}. Reply as
   the next speaker in the conversation only based on the
   conversation history and the subtext thought. Format your reply
   as {Speaker X:message}. Do not say you are a chatbot. Do not
   output any further thoughts."
prompt2A="You will be given an excerpt of a conversation between two
    speakers. It will be formatted as {Speaker 1: message, Speaker
```

```
    2: message}, followed by {Thought:}. Reply as the next speaker in
     the conversation only based on the conversation history and the
    subtext thought of the speaker replying. Format your reply as {
    Speaker X:message}. Do not say you are a chatbot. Do not output
    any further thoughts."

#systemPromptRigged="You will be given an excerpt of a conversation
    between two speakers. Format your output as {Output:}. Your task
    is to generate the subtext thought that the next speaker is
    having, based on the following idea:"
promptVotingRiggedChoice="You are a cognitive scientist. Based on
    the conversation history, which of these subtexts was the last
    speaker most likely thinking before they spoke? Choose the one
    that best fits the context. Output a single number corresponding
    to the thought you choose. Output this number as {Output:num}. "
promptScoringRiggedScoring="You are a cognitive scientist. Based on
    the conversation history and the given subtext, rate how likely
    it is that this subtext reflects what the last speaker was
    thinking before they spoke. Output a score from 1 (very unlikely)
     to 10 (very likely) as {Output:num}"


all_subtexts_parallel_list=[]# a list of length s (number of
    subtexts) containing lists of length n (number of steps)
all_subtexts_parallel_list=[prompt1,prompt2A]
all_subtexts_random_parallel_list=[prompt1Random,prompt2A]


#[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (
    generate next speaker sentence)]
prompts_subtext_voting1=[prompt1,promptVoting,prompt2A]#choice
```

```
prompts_subtext_voting2=[prompt1,promptVotingScoring,prompt2A]#scoring

prompts_subtext_voting3=[prompt1,promptVotingRiggedChoice,prompt2A]#voting_choice_rigged
prompts_subtext_voting4=[prompt1,promptScoringRiggedScoring,prompt2A]#voting_scoring_rigged


myDict = {}
myDict["prompt_singleStep"] = "You will be given an excerpt of a conversation that includes the next speaker. It will be formatted as {Speaker 1: message, Speaker 2: message}. Reply as the next speaker in the conversation only. Format your reply as {Speaker 1: message}. Do not say you are a chatbot."
myDict["prompts_subtext_multiStep"] = all_subtexts_parallel_list
myDict["prompts_subtext_multiStep_random"] = all_subtexts_random_parallel_list
myDict["subtexts"] = selectedSubtexts
myDict["prompts_subtext_voting1"]=prompts_subtext_voting1
myDict["prompts_subtext_voting2"]=prompts_subtext_voting2
myDict["randomSubtextList"] = randomSubtextList
myDict["prompts_subtext_voting3"] = prompts_subtext_voting3
myDict["prompts_subtext_voting4"]=prompts_subtext_voting4


exportJSONFile(myDict,"prompts/promptsC_only4.json")

promptOld="You will be given an excerpt of a conversation from the point of view of one speaker. Your task is to reply as the other speaker in the conversation. Reply as a human in the conversation not a chatbot."
```

```



#instead we want prompts to be structured like this:
#[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (
    generate next speaker sentence)]
}
```

A.3 Dataset filtering and setup

```
import pandas as pd
import numpy as np
import json
import matplotlib.pyplot as plt
from datasetUtilities import createDataset,exportJSONFile
import csv

df = pd.read_csv('decoding_df_798_final_notasks.csv')
print(df.columns)


sentence=df["sentence"]
corrupted=df["corrupted"]
speaker=df["speaker"]
conv_name=df["conversation_name"]



onsets=df["onsets"]
offsets=df["offsets"]

gap_time = np.zeros(len(onsets))  # Convert gap_time to a NumPy
```

```
    array
for i in range(len(onsets)):
    if i == 0:
        gap_time[i] = 0  # or some default value
    else:
        gap_time[i] = onsets[i] - offsets[i - 1]

print(gap_time[0:5])
print(len(gap_time))
N=len(speaker)#length of dataset
switchConvo = np.zeros(N, dtype=object)


#segmenting data into "two speaker segments", and also segmenting
    out corrupted segments
#get first two unique speakers
lastTwoSpeakers, index=np.unique(speaker,return_index=True)
lastTwoSpeakers=lastTwoSpeakers[np.argsort(index)]
lastTwoSpeakers=list(lastTwoSpeakers)[:2]

print(lastTwoSpeakers)

for i in range(len(sentence)):
    curSpeaker=speaker[i]
    #keep track of last two speakers, add new segment if one of them
     changes and update lastTwoSpeakers
    if (curSpeaker not in lastTwoSpeakers):
        prevSpeaker=speaker[i-1]
        lastTwoSpeakers.clear()
        lastTwoSpeakers.append(prevSpeaker)
        lastTwoSpeakers.append(curSpeaker)
        switchConvo[i]=1
```

```
        #print("switch speaker" + str(i))
        #print(curSpeaker)
        #print(lastTwoSpeakers)

    #segment out corrupted data
    if (corrupted[i]==1):
        switchConvo[i]=1
    if (i != 0):
        if (corrupted[i-1]==1):
            switchConvo[i]=1
        if (conv_name[i]!=conv_name[i-1]):#add new segment if conversation as labelled in data changes
            switchConvo[i]=1


#all the indicies where a new segment starts
start_indices = np.where(switchConvo == 1)[0]
start_indices =np.insert(start_indices,0,0)


# Create masks for each segment
masks = []
for i, start in enumerate(start_indices):
    # Determine the end of the segment (either next start index or end of array)
    end = start_indices[i+1] if i+1 < len(start_indices) else len(switchConvo)
    # Create a mask for the segment
    mask = np.zeros_like(switchConvo, dtype=bool)
    mask[start:end] = True

    #remove corrupted segments
```

```
    if (corrupted[start]==False):
        masks.append(mask)


#segmenting data into segments based on masks
sentenceSegments = [sentence[mask] for mask in masks] # masked sentences
speakerSegments = [speaker[mask] for mask in masks] # masked speaker labels
gap_timeSegments = [gap_time[mask] for mask in masks] # masked gap times
overallConversation=[]
overallGapTimes=[]

#checking for corrupted segments
print("Checking for corrupted segments")
corruptedSegments = [corrupted[mask] for mask in masks]
for i in range(len(corruptedSegments)):
    if (corruptedSegments[i].any()):
        print("missed corrupted segment!!!")

#finding the percentage of the sentences spoken by the speaker that talks the most in each conversation
percentageSpeak=[]
for i in range(len(speakerSegments)):
    percentageSpeak.append(speakerSegments[i].value_counts(normalize=True).max())

plt.figure(figsize=(10, 6))
plt.hist(percentageSpeak, bins=30, edgecolor='black')
plt.xlabel('Percentage Spoken')
plt.ylabel('Frequency')
```

```
plt.title('Histogram of percentageSpeak-percentage of sentences
    spoken by the speaker that talks the most in each conversation')
plt.show()
plt.savefig('percentageSpeak.png')

#joining sentences from each speaker into conversational turns, and
    formatting data to input to createDataset function
for i in range(len(sentenceSegments)):
    #if conversation dominated by one speaker skip it
    if (speakerSegments[i].value_counts(normalize=True).max() >
    0.75):
        continue

    sentenceSegment=list(sentenceSegments[i])
    speakerSegment=list(speakerSegments[i])
    gap_timeSegment=list(gap_timeSegments[i])

    prevSpeaker=speakerSegment[0]#get initial speaker
    curUtterance=""

    conversationHistory=[]
    gapTimeHistory=[]
    for j in range(len(sentenceSegment)):
        #if speaker changes, add utterance so far from prev speaker
    to conversation history
        if (prevSpeaker!=speakerSegment[j]):
            conversationHistory.append(curUtterance)
            curUtterance=""
            curGapTime=gap_timeSegment[j]
            gapTimeHistory.append(curGapTime)
        #add next sentence to current uttranace (add a space if not
    the first sentence)
```

```
        if j!=0:
            curUtterance=curUtterance+ " " + sentenceSegment[j]
        else:
            curUtterance=sentenceSegment[j]
            gapTimeHistory.append(0)#add 0 gap time for first sentence


        #updating the previous speaker
        prevSpeaker=speakerSegment[j]
    if gapTimeHistory:
        gapTimeHistory.pop()  # Remove the last entry of gapTimeHistory
    overallConversation.append(conversationHistory)
    overallGapTimes.append(gapTimeHistory)





print(len(overallGapTimes[0]))
print(len(overallConversation[0]))

#shows how many conversations were filtered out
print("Num conversations before:")
print(len(sentenceSegments))
print("Num conversations after:")
print(len(overallConversation))

#plotting number of conversational turns in each conversation
conversationalTurns=[]
for i in overallConversation:
```

```
    conversationalTurns.append(len(i))



plt.figure(figsize=(10, 6))
plt.hist(conversationalTurns, bins=30, edgecolor='black')
plt.xlabel('Number of conversational turns')
plt.ylabel('Frequency')
plt.title('Histogram of number of conversational turns in each
    conversation')
plt.show()
plt.savefig('conversationLengths.png')

print("Average number of conversational turns:", np.mean(
    conversationalTurns))


#creating correctly formatted datasets
references=list()
labels=list()
references2=list()
labels2=list()

max_num_turns=15

references,labels=createDataset(overallConversation,1,max_num_turns
    -1)
references2,labels2=createDataset(overallConversation,2,
    max_num_turns-2)
references3,labels3=createDataset(overallConversation,3,
    max_num_turns-3)
references5,labels5=createDataset(overallConversation,5,
```

```
    max_num_turns-5)
references10,labels10=createDataset(overallConversation,10,max_num_turns-10)
references15,labels15=createDataset(overallConversation,15,max_num_turns-15)

referencesGapTimes15,labelsGapTimes15=createDataset(overallGapTimes,15,max_num_turns-15)


references3_extra,labels3_extra=createDataset(overallConversation,3,3-3)
referencesGapTimes3_extra,labelsGapTimes3_extra=createDataset(overallGapTimes,3,3-3)

print("Number of label sentences: " + str(len(labels)))
print("labels identical:")
print(labels10==labels15)
print(labels5==labels10)
print(labels3==labels5)
print(labels2==labels3)
print(labels==labels2)

exportJSONFile(references,labels,"lab_data_no_tasks_one_turn.json")
exportJSONFile(references2,labels2,"lab_data_no_tasks_two_turn.json")
exportJSONFile(references3,labels3,"lab_data_no_tasks_three_turn.json")
exportJSONFile(references5,labels5,"lab_data_no_tasks_five_turn.json")
exportJSONFile(references10,labels10,"lab_data_no_tasks_ten_turn.json")
```

```
exportJSONFile(references15,labels15,"lab_data_no_tasks_fifteen_turn
    .json")

exportJSONFile(referencesGapTimes15,labelsGapTimes15,"
    lab_data_no_tasks_gap_times_fifteen_turn.json")

exportJSONFile(references3_extra,labels3_extra,"
    lab_data_no_tasks_three_turn_extra.json")
exportJSONFile(referencesGapTimes3_extra,labelsGapTimes3_extra,"
    lab_data_no_tasks_gap_times_three_turn_extra.json")

print(len(labelsGapTimes3_extra))
```

```
    import json

#formats the dataset for inference
#input is structured by conversation and then conversational turns
#turnCount is num of prev. conversational turns to include
#startPoint is used to make sure different datasets have the same
    references e.g. if startPoint=0, one turn would have more data
    than two turn (one sentence for each conversation)
def createDataset(dataset,turnCount,startPoint=0):
    _prompts=list()
    _references=list()

    #for each conversation
    for i in range (len(dataset)):
        excerpt=dataset[i]
        #for all sentence sequences we can use
        if len(excerpt)<turnCount+1:
            continue

        for j in range (startPoint,len(excerpt)-turnCount):
```

```python

            prompt=""

            #add current speaker to prompt string (assumes alternating speakers)
            for k in range(0,turnCount):
                speakerNum=k %2 + 1

                prompt = prompt + "Speaker " + str(speakerNum) + ": " + str(excerpt[j+k])

                if k!=turnCount-1:
                    prompt=prompt + " "

            #use next utterance as the reference
            reference=excerpt[j+turnCount]

            _prompts.append(prompt)
            _references.append(reference)

    return _prompts, _references


def exportJSONFile(_prompts,_references,fileName):
    myDict = {}
    myDict["references"] = _prompts
    myDict["labels"] = _references

    json_object = json.dumps(myDict)

    with open(fileName, "w") as outfile:
        outfile.write(json_object)
```

A.4 Inference generation

```python

import torch
import time, subprocess, argparse, json, re, math
import accelerate.utils
import numpy as np
import pandas as pd
from transformers import AutoModelForCausalLM, AutoTokenizer,
    AutoModel
from evaluate import load
from pathlib import Path
from datetime import datetime
import os
import math
import random
from inference import generateInference
from distutils.util import strtobool

def str2bool(v):
    return bool(strtobool(v))  # Converts "true"/"false", "yes"/"no
    ", "1"/"0" to boolean


numSamplesTotal=200



def getSimilarityResults(_references,_labels, sys_prompt, model,
    tokenizer, bertscore, modelType="Llama8B", bertscore_model="
    distilbert-base-uncased",_temperature=1.0):
    #perform inference and get similarity results through BERTscore

```

```python
    #SET UP PROMPTS
    #Other prompt: You will be given an excerpt of a conversation from the point of view of one speaker. Your task is to reply as the other speaker in the conversation. Reply as a human in the conversation not a chatbot.
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": sys_prompt}, {"role": "user", "content": _references[i]}] )
    #getting next sentence and outputting similarity results
    gen_text=generateInference(myinput, model, tokenizer, modelType, _temperature=_temperature)
    print("Generated text: with tempeature " + str(_temperature))
    results = bertscore.compute(predictions=gen_text, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text

def getSimilarityResultsMultiStep(_references,_labels, sys_prompt, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased"):
    #perform inference and get similarity results through BERTscore

    #SET UP PROMPTS
    prompt1=sys_prompt[0]
    prompt2=sys_prompt[1]

    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt1}, {"role": "user", "content": _references[i]}] )

```

```
    gen_text1=generateInference(myinput, model, tokenizer, modelType)#subtext thought
    gen_text1 = [re.sub("Thought:", "",i,3) for idx, i in enumerate(gen_text1)]

    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt2}, {"role": "user", "content": _references[i]  + "Thought: " + gen_text1[i]   }  ])


    gen_text2=generateInference(myinput, model, tokenizer, modelType)#final output (generated speaker sentence(s))
    gen_text2 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text2)]

    results = bertscore.compute(predictions=gen_text2, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text1, gen_text2

def getSimilarityResultsMultiStepRandomThought(_references,_labels, sys_prompt, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased",random_thought_list=None):
    #perform inference and get similarity results through BERTscore

    #SET UP PROMPTS
    prompt1=sys_prompt[0]
    prompt2=sys_prompt[1]

```

```python
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt1}, {"role": "user", "content": _references[i]}] )

    #gen_text1 = ["no actual thought"] * len(_references)  # Create an empty array of the same length as _references

    if random_thought_list is not None:
        if len(random_thought_list) >= len(_references):
            gen_text1 = random.sample(random_thought_list, k=len(_references))
        else:
            gen_text1 = random.choices(random_thought_list, k=len(_references))
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt2}, {"role": "user", "content": _references[i]  + "Thought: " + gen_text1[i]   }  ])


    gen_text2=generateInference(myinput, model, tokenizer, modelType)#final output (generated speaker sentence(s))
    gen_text2 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text2)]

    #gen_text2=gen_text1
    results = bertscore.compute(predictions=gen_text2, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text1, gen_text2
```

```


def getSimilarityResultsVoteChoice(_references,_labels, sys_prompt, subtexts, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased"):
    #perform inference and get similarity results through BERTscore
    ##[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (generate next speaker sentence)]

    #SET UP PROMPTS
    prompt1=sys_prompt[0]  #list of subtext prompts
    prompt2=sys_prompt[1] # voting prompt
    prompt3=sys_prompt[2] # generate next speaker sentence

    #generating subtext thoughts
    #want to keep the references size, so generate along the other column and then reshape the array
    gen_text_thoughts=[]
    for i in range (0,len(prompt1)):
        myinput=[]
        for j in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt1[i]}, {"role": "user", "content": _references[j]}] )
        gen_text1=generateInference(myinput, model, tokenizer, modelType)#subtext thought
        gen_text1 = [re.sub("Thought:", "",i,3) for idx, i in enumerate(gen_text1)]
        gen_text_thoughts.append(gen_text1)

    # Transpose the list of lists
    gen_text_thoughts = list(map(list, zip(*gen_text_thoughts))) #output should be a list of lists (x is reference, y is different
```

```
subtext prompts)
    print("Gen thoughts shape:")
    print(len(gen_text_thoughts))
    print(len(gen_text_thoughts[0]))
    #voting

    for i in range(len(gen_text_thoughts)):#this should be length references
        #create the list of sutexts to choose from
        thoughtList=[]
        thoughtPrompt = "\n".join([f"{idx+1}. {thought}" for idx, thought in enumerate(subtexts)])

        myinput=[]
        for i in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt2}, {"role": "user", "content": "Conversation History: " + _references[i] + "Thought options: " + thoughtPrompt  }  ])

    gen_text2=generateInference(myinput, model, tokenizer, modelType,_temperature=0.0)#result of voting (should just be a single number)

    #need to select which one was voted for
    voted_thought=[]
    numErrors=0
    for i in range (len(gen_text_thoughts)):
        parsedString=None
        match = re.search(r'Output:\s*(\d+)', gen_text2[i])
        if match:
            parsedString = match.group(1)
        else:
```

```
         match = re.search(r'\d+', gen_text2[i])
         if match:
             parsedString = match.group(0)
         else:
             print("Error: no number found in the output: " + gen_text2[i])
             parsedString = "1"  # default to "1" if no match is found
             numErrors+=1

     if int(parsedString) > len(gen_text_thoughts[i]):
         print("Error: number found is greater than the number of thoughts. The gen_text was: " + gen_text2[i])
         parsedString = "1"
     if int(parsedString) < 1:
         print("Error: number found is less than 1, which is not a valid option. The gen_text was: " + gen_text2[i])
         parsedString = "1"
     voted_thought.append(int(parsedString) - 1)

 print("Number of errors in voting in this batch: " + str(numErrors))
 #need to generate the next speaker sentence  based on thought like before

 myinput=[]
 for i in range (0,len(_references)):
     myinput.append([{"role": "system", "content": prompt3}, {"role": "user", "content": _references[i]  + "Thought: " + gen_text_thoughts[i][voted_thought[i]]  }  ])

 gen_text3=generateInference(myinput, model, tokenizer, modelType
```

```
)#final output (generated speaker sentence(s))
    gen_text3 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text3)]

    results = bertscore.compute(predictions=gen_text3, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text_thoughts, gen_text2, gen_text3, voted_thought

def getSimilarityResultsVoteScoring(_references,_labels, sys_prompt, subtexts, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased"):
    #perform inference and get similarity results through BERTscore
    ##[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (generate next speaker sentence)]

    #SET UP PROMPTS
    prompt1=sys_prompt[0]  #list of subtext prompts
    prompt2=sys_prompt[1] # scoring prompt
    prompt3=sys_prompt[2] # generate next speaker sentence

    #generating subtext thoughts and then scoring them
    #want to keep the references size, so generate along the other column and then reshape the array
    gen_text_thoughts=[]
    for i in range (0,len(prompt1)):
        myinput=[]
        for j in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt1[i]}, {"role": "user", "content": _references[j]}] )
        gen_text1=generateInference(myinput, model, tokenizer,
```

```
modelType)#subtext thought
        gen_text1 = [re.sub("Thought:", "",i,3) for idx, i in enumerate(gen_text1)]
        gen_text_thoughts.append(gen_text1)
    gen_text_thoughts = list(map(list, zip(*gen_text_thoughts)))  # Transpose the list of lists, output should be a list of lists (x is reference, y is different subtext prompts)
    print("Gen thoughts shape:")
    print(len(gen_text_thoughts))
    print(len(gen_text_thoughts[0]))

    #voting - instead of inputting the whole thought options, evaluate all the thought options separately with a probability/score, parse the score and then choose the vote with the highest score

    #generating scores for each subtext
    gen_text_scores=[]
    for i in range (0,len(prompt1)):
        myinput=[]
        for j in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt2}, {"role": "user", "content": "Conversation History: " + _references[j] + "Subtext: " + subtexts[i] }] )
        gen_text2=generateInference(myinput, model, tokenizer, modelType,_temperature=0.0)
        gen_text_scores.append(gen_text2)

    gen_text_scores = list(map(list, zip(*gen_text_scores)))      # Transpose the list of lists, output should be a list of lists (x is reference, y is different subtext prompts)
    print("Gen scores shape:")
```

```
    print(len(gen_text_scores))
    print(len(gen_text_scores[0]))

    #parsing the list of scores to get the best thought for each reference
    all_subtext_scores=[]#contains all the scores for all references for each subtext
    best_score_values=[]
    best_scored_thoughts=[0]*len(gen_text_scores)
    best_score_indexes=[]

    numErrors=0
    for i in range(len(gen_text_scores)):#this should be length references
        subtext_scores=[]

        for j in range(len(gen_text_scores[i])):#this should be length subtexts
            parsedString=None
            match=None
            match = re.search(r'Output:\s*(\d+)', gen_text_scores[i][j])
            if match:
                parsedString = match.group(1)
            else:
                match = re.search(r'\d+', gen_text_scores[i][j])
                if match:
                    parsedString = match.group(0)
                else:
                    print("Error: no number found in the output: " + gen_text_scores[i][j])
                    parsedString = "1"  # default to "1" if no match
```

```
is found
                numErrors+=1

        subtext_scores.append(int(parsedString))


    #get the max of the subtext scores
    max_score=max(subtext_scores)
    max_score_index=subtext_scores.index(max_score)

    all_subtext_scores.append(subtext_scores)
    best_score_values.append(max_score)
    best_scored_thoughts[i]=gen_text_scores[i][max_score_index]
    best_score_indexes.append(max_score_index)

print("Number of errors in parsing scores in this batch: " + str(numErrors))


#need to generate the next speaker sentence on the thought that is evaluated to be the best one
myinput=[]
for i in range (0,len(_references)):
     myinput.append([{"role": "system", "content": prompt3}, {"role": "user", "content": "Conversation History: " + _references[i] + "Thought: " + best_scored_thoughts[i] }] )

gen_text3=generateInference(myinput, model, tokenizer, modelType)#final output (generated speaker sentence(s))
gen_text3 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text3)]

```

```python
    print(len(gen_text3))
    results = bertscore.compute(predictions=gen_text3, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text_thoughts, gen_text_scores, gen_text3, best_score_indexes, all_subtext_scores, best_score_values

def getSimilarityResultsVoteChoiceRigged(_references,_labels, sys_prompt, subtexts, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased"):
    #perform inference and get similarity results through BERTscore
    ##[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (generate next speaker sentence)]

    #SET UP PROMPTS
    prompt1=sys_prompt[0]  #list of generation of thought prompts inc. subtexts
    prompt2=sys_prompt[1] # voting prompt (rigged)
    prompt3=sys_prompt[2] # generate next speaker sentence

    for i in range(len(_references)):#this should be length references
        #create the list of subtexts to choose from
        thoughtList=[]
        thoughtPrompt = "\n".join([f"{idx+1}. {thought}" for idx, thought in enumerate(subtexts)])

        myinput=[]
        for i in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt2}, {"role": "user", "content": "Conversation History: " +
```

```
_references[i] + "Last Speaker: " + _labels[i] + "Thought options: " + thoughtPrompt  }  ])

    gen_text2=generateInference(myinput, model, tokenizer, modelType,_temperature=0.0)#result of voting (should just be a single number)

    #need to select which one was voted for
    voted_thought=[]
    numErrors=0
    for i in range (len(_references)):
        parsedString=None
        match = re.search(r'Output:\s*(\d+)', gen_text2[i])
        if match:
            parsedString = match.group(1)
        else:
            match = re.search(r'\d+', gen_text2[i])
            if match:
                parsedString = match.group(0)
            else:
                print("Error: no number found in the output: " + gen_text2[i])
                parsedString = "1"  # default to "1" if no match is found
                numErrors+=1

        if int(parsedString) > len(prompt1):
            print("Error: number found is greater than the number of thoughts. The gen_text was: " + gen_text2[i])
            parsedString = "1"
        if int(parsedString) < 1:
            print("Error: number found is less than 1, which is not
```

```
a valid option. The gen_text was: " + gen_text2[i])
         parsedString = "1"
     voted_thought.append(int(parsedString) - 1)

 print("Number of errors in voting in this batch: " + str(numErrors))
 #need to generate the next speaker sentence  based on thought like before

 # Generate thoughts based on voted_thought directly in batches
 gen_text_thoughts = [["no thoughts"] * len(sys_prompt[0]) for _ in range(len(_references))]
 batch_inputs = []
 for i in range(len(_references)):
     voted_thought_index = voted_thought[i]
     batch_inputs.append([{"role": "system", "content": sys_prompt[0][voted_thought_index]},
                          {"role": "user", "content": _references[i]}])

 # Perform batch inference
 batch_gen_texts = generateInference(batch_inputs, model, tokenizer, modelType)

 # Update gen_text_thoughts with the generated results
 for i, gen_text in enumerate(batch_gen_texts):
     #print(gen_text)
     voted_thought_index = voted_thought[i]
     gen_text_thoughts[i][voted_thought_index] = re.sub("Thought: ", "", gen_text, 3)
 print("Generated thoughts based on voted_thought.")

```

```
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt3}, {"role": "user", "content": _references[i]  + "Thought: " + gen_text_thoughts[i][voted_thought[i]]  }  ])

    gen_text3=generateInference(myinput, model, tokenizer, modelType)#final output (generated speaker sentence(s))
    gen_text3 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text3)]

    results = bertscore.compute(predictions=gen_text3, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text_thoughts, gen_text2, gen_text3, voted_thought


def getSimilarityResultsVoteScoringRigged(_references,_labels, sys_prompt,subtexts, model, tokenizer, bertscore, modelType="Llama8B", bertscore_model="distilbert-base-uncased"):
    #perform inference and get similarity results through BERTscore
    ##[[prompt1 (generate subtext)*10], prompt2 (voting), prompt3 (generate next speaker sentence)]

    #SET UP PROMPTS
    prompt1=sys_prompt[0]  #list of subtext prompts
    prompt2=sys_prompt[1] # scoring prompt
    prompt3=sys_prompt[2] # generate next speaker sentence


```

```
    #voting - instead of inputting the whole thought options, evaluate all the thought options separately with a probability/score, parse the score and then choose the vote with the highest score

    #generating scores for each subtext
    gen_text_scores=[]
    for i in range (0,len(prompt1)):
        myinput=[]
        for j in range (0,len(_references)):
            myinput.append([{"role": "system", "content": prompt2},{"role": "user", "content": "Conversation History: " + _references[j] + ". Last Speaker: " + _labels[j] + ". Subtext Thought: " + subtexts[i] }] )
        gen_text2=generateInference(myinput, model, tokenizer, modelType,_temperature=0.0)
        gen_text_scores.append(gen_text2)

    gen_text_scores = list(map(list, zip(*gen_text_scores)))      # Transpose the list of lists, output should be a list of lists (x is reference, y is different subtext prompts)
    print("Gen scores shape:")
    print(len(gen_text_scores))
    print(len(gen_text_scores[0]))

    #parsing the list of scores to get the best thought for each reference
    all_subtext_scores=[]#contains all the scores for all references for each subtext
    best_score_values=[]
    best_scored_thoughts=[0]*len(gen_text_scores)
    best_score_indexes=[]
```

```
 numErrors=0
 for i in range(len(gen_text_scores)):#this should be length references
     subtext_scores=[]

     for j in range(len(gen_text_scores[i])):#this should be length subtexts
         parsedString=None
         match=None
         match = re.search(r'Output:\s*(\d+)', gen_text_scores[i][j])
         if match:
             parsedString = match.group(1)
         else:
             match = re.search(r'\d+', gen_text_scores[i][j])
             if match:
                 parsedString = match.group(0)
             else:
                 print("Error: no number found in the output: " + gen_text_scores[i][j])
                 parsedString = "1"  # default to "1" if no match is found
                 numErrors+=1

         subtext_scores.append(int(parsedString))


     #get the max of the subtext scores
     max_score=max(subtext_scores)
     max_score_index=subtext_scores.index(max_score)

```

```
        all_subtext_scores.append(subtext_scores)
        best_score_values.append(max_score)
        best_scored_thoughts[i]=gen_text_scores[i][max_score_index]
        best_score_indexes.append(max_score_index)

    print("Number of errors in parsing scores in this batch: " + str(numErrors))

    # Generate thoughts based on voted_thought directly
     # Generate thoughts based on voted_thought directly in batches
    gen_text_thoughts = [["no thoughts"] * len(sys_prompt[0]) for _ in range(len(_references))]
    batch_inputs = []
    for i in range(len(_references)):
        voted_thought_index = int(best_score_indexes[i])  # Ensure it's an integer
        batch_inputs.append([{"role": "system", "content": sys_prompt[0][voted_thought_index]},
                             {"role": "user", "content": _references[i]}])

    # Perform batch inference
    batch_gen_texts = generateInference(batch_inputs, model, tokenizer, modelType)

    # Update gen_text_thoughts with the generated results
    for i, gen_text in enumerate(batch_gen_texts):
        #print(gen_text)
        voted_thought_index = best_score_indexes[i]
        gen_text_thoughts[i][voted_thought_index] = re.sub("Thought: ", "", gen_text, 3)
    print("Generated thoughts based on voted_thought.")
```

```



    #need to generate the next speaker sentence on the thought that is evaluated to be the best one
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": prompt3}, {"role": "user", "content": "Conversation History: " + _references[i] + "Thought: " + gen_text_thoughts[i][best_score_indexes[i]] }])

    gen_text3=generateInference(myinput, model, tokenizer, modelType)#final output (generated speaker sentence(s))
    gen_text3 = [re.sub("Speaker [0-9]:", "",i,3) for idx, i in enumerate(gen_text3)]

    print(len(gen_text3))
    results = bertscore.compute(predictions=gen_text3, references=_labels, model_type=bertscore_model)

    return results['f1'], gen_text_thoughts, gen_text_scores, gen_text3, best_score_indexes, all_subtext_scores, best_score_values



#want this to be the ame for all prompting types
def performAnalysisOnWholeData(references,labels, sys_prompt, prompting_type, model_name, model, tokenizer, bertscore, shuffled_indices,subtexts=None,_temperature=1.0, random_thought_list=None):
```

```
    batchSize=20
    batchNum=math.ceil(len(references)/batchSize)


    results_dict = {
        "reference": references,
        "label": labels,
    }

    if prompting_type != "multi_parallel":
        results_dict["gen_speech"]= []
        results_dict["bertscore"]= []

    if prompting_type == "voting_score" or prompting_type == "voting_choice" or prompting_type == "voting_choice_rigged" or prompting_type == "voting_score_rigged":
        results_dict["gens_thoughts"] = []
        results_dict["gens_vote"] = []
        results_dict["voted_thoughts_parsed"] = []
        results_dict["voted_thoughts_true"] = []
        results_dict["gen_for_voted_thought"] = []
        results_dict["subtext_for_voted_thoughts"] = []

    for i in range(batchNum):
        start=batchSize*i
        end = len(references) if i == batchNum - 1 else batchSize * (i + 1)
        if prompting_type == "single":
            batch_scores, batch_gens_speech = (
                getSimilarityResults(references[start:end], labels[start:end], sys_prompt, model, tokenizer, bertscore, model_name, _temperature=_temperature)
```

```
            )
        if prompting_type == "multi_parallel":
            for p in range(len(sys_prompt[0])):
                print(f"Running for subtext {p}")
                batch_prompt=[sys_prompt[0][p],sys_prompt[1]]

                if random_thought_list is None:
                    batch_scores, batch_gens_thought, batch_gens_speech = (
                        getSimilarityResultsMultiStep(references[start:end], labels[start:end], batch_prompt, model, tokenizer, bertscore, model_name)
                    )
                else:
                    batch_scores, batch_gens_thought, batch_gens_speech = (
                        getSimilarityResultsMultiStepRandomThought(references[start:end], labels[start:end], batch_prompt, model, tokenizer, bertscore, model_name, random_thought_list=random_thought_list)
                    )
                results_dict.setdefault(f"subtext{p}", []).extend([subtexts[p]]*(end-start))#may need to expand this by batch size
                results_dict.setdefault(f"gen_thought{p}", []).extend(batch_gens_thought)
                results_dict.setdefault(f"gen_speech{p}", []).extend(batch_gens_speech)
                results_dict.setdefault(f"bertscore{p}", []).extend(batch_scores)

        elif prompting_type == "voting_score":
            batch_scores, batch_gens_thought, batch_gens_vote,
```

```
batch_gens_speech, batch_voted_thought, batch_all_subtext_scores, batch_best_score_values = (
                getSimilarityResultsVoteScoring(references[start:end], labels[start:end], sys_prompt, subtexts, model, tokenizer, bertscore, model_name)
            )
        elif prompting_type == "voting_choice":
            batch_scores, batch_gens_thought, batch_gens_vote, batch_gens_speech, batch_voted_thought = (
                getSimilarityResultsVoteChoice(references[start:end], labels[start:end], sys_prompt, model, tokenizer, bertscore, model_name)
            )
        elif prompting_type == "voting_choice_rigged":#
            batch_scores, batch_gens_thought, batch_gens_vote, batch_gens_speech, batch_voted_thought = (
                getSimilarityResultsVoteChoiceRigged(references[start:end], labels[start:end], sys_prompt, subtexts, model, tokenizer, bertscore, model_name)
            )
        elif prompting_type == "voting_score_rigged":
            batch_scores, batch_gens_thought, batch_gens_vote, batch_gens_speech, batch_voted_thought, batch_all_subtext_scores, batch_best_score_values = (
                getSimilarityResultsVoteScoringRigged(references[start:end], labels[start:end], sys_prompt, subtexts, model, tokenizer, bertscore, model_name)
            )

        if prompting_type == "voting_choice" or prompting_type == "voting_score" or prompting_type == "voting_choice_rigged" or prompting_type == "voting_score_rigged":
```

```
        voted_thoughts_strings = [batch_gens_thought[i][voted_thought] for i, voted_thought in enumerate(batch_voted_thought)]
        voted_thoughts_subtext = [subtexts[i] for i in batch_voted_thought]

        results_dict.setdefault("gens_thoughts", []).extend(batch_gens_thought)
        results_dict.setdefault("gens_vote", []).extend(batch_gens_vote)#for scoring choice this is the list of generated scores
        if prompting_type=="voting_score" or prompting_type=="voting_score_rigged":
            results_dict.setdefault("gens_vote_unshuffled", []).extend(unshuffleArray(batch_gens_vote, shuffled_indices))
        results_dict.setdefault("voted_thoughts_parsed", []).extend(batch_voted_thought)

        # Undo the shuffling for subtexts, gens_thoughts, voted_thoughts
        reverse_mapping =shuffled_indices             #map from order in shuffled array to order in original array
        # Apply the reverse mapping to restore the original order
        test_array=[shuffled_indices for _ in range(len(batch_gens_thought))]
        unshuffled_subtexts=unshuffleArray(test_array, shuffled_indices)
        unshuffled_gens_thoughts=unshuffleArray(batch_gens_thought, shuffled_indices)
        voted_thoughts_true = [reverse_mapping[batch_voted_thought[i]] for i in range(len(batch_voted_thought))]
```

```


            results_dict.setdefault("voted_thoughts_true", []).extend(voted_thoughts_true)
            results_dict.setdefault("gen_for_voted_thought", []).extend(voted_thoughts_strings)
            results_dict.setdefault("subtext_for_voted_thoughts", []).extend(voted_thoughts_subtext)
            results_dict.setdefault("gens_thoughts_unshuffled", []).extend(unshuffled_gens_thoughts)
            results_dict.setdefault("unshuffled_subtexts_indices", []).extend(unshuffled_subtexts)
            if prompting_type == "voting_score" or prompting_type == "voting_score_rigged":#the part that's unique to voting_scores
                results_dict.setdefault("all_subtext_scores_unshuffled", []).extend(unshuffleArray(batch_all_subtext_scores, shuffled_indices))
                results_dict.setdefault("best_score_values", []).extend(batch_best_score_values)
        if prompting_type != "multi_parallel":
            results_dict["bertscore"].extend(batch_scores)
            results_dict["gen_speech"].extend(batch_gens_speech)


    for key, value in results_dict.items():
        print(f"{key}: {len(value)}")

    if "results" in results_dict:
        results = results_dict["results"]
        print("Mean result: " + str(np.mean(results)))
        print("Standard deviation results: " + str(np.std(results)))
        print("Number of samples: " + str(len(results)))
```

```
        print("Std deviation of mean: " + str(np.std(results) / np.sqrt(len(results))))

    return results_dict


def unshuffleArray(gens_thoughts, shuffled_indices):
    temp_array1=[]
    for i in range(len(gens_thoughts)):
        temp_array2=[]
        for j in range(len(shuffled_indices)):
            index_in_shuffled = shuffled_indices.index(j)
            temp_array2.append(gens_thoughts[i][index_in_shuffled])
        temp_array1.append(temp_array2)
    gens_thoughts=temp_array1
    return gens_thoughts


def main():
    parser = argparse.ArgumentParser(description="Run analysis with specified model and data.")
    parser.add_argument('--model_name', type=str, required=True, help='Name of the LM to use')
    parser.add_argument('--data_path', type=str, required=True, help='Path to the data file')
    parser.add_argument('--prompt_path', type=str, required=True, help='Path to the prompt file')
    parser.add_argument('--num_shuffles', type=int, required=True, help='Number of random shuffles to do (first set is always not shuffled)')
    parser.add_argument('--prompting_type', type=str, required=True, choices=["single","multi_parallel","voting_choice","voting_score
```

```
","voting_choice_rigged","voting_score_rigged"], help='Prompting type: standard or voting')
    parser.add_argument('--temperature', type=float, required=False, help='Temperature of generation sampling', default=1.0)
    parser.add_argument('--random_subtext', type=str, required=False, help='Whether a set of random subtexts are used', default="False")
    parser.add_argument('--random_thought', type=str, required=False, help='Whether a set of random thoughts are used', default="False")

    args = parser.parse_args()

    model_name = args.model_name
    data_path = Path(args.data_path)
    prompt_path = Path(args.prompt_path)
    num_shuffles = args.num_shuffles
    random_shuffle = num_shuffles > 1 #if num_shuffles is 1, don't shuffle
    prompting_type = args.prompting_type
    _temperature = args.temperature
    random_subtext = str2bool(args.random_subtext)
    random_thought = str2bool(args.random_thought)


    bertscore = load("bertscore")
    #set up device to GPU if available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")


```

```python
    #LOADING AND SETTING UP THE MODEL
    #set up model id based on model type

    if (model_name=="Llama1B"):
        model_id = "meta-llama/Llama-3.2-1B-Instruct"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.bfloat16, device_map="auto").to(device)
    elif (model_name=="Llama8B"):
        model_id = "meta-llama/Llama-3.1-8B-Instruct"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.bfloat16, device_map="auto").to(device)
    elif (model_name=="Llama70B"):
        model_id ="hugging-quants/Meta-Llama-3.1-70B-Instruct-GPTQ-INT4"
        #model_id = "unsloth/Meta-Llama-3.1-70B-Instruct-bnb-4bit"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16).to(device)
        # model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto", cache_dir='/scratch/gpfs/ij9216/cached_models').to(device)
        # model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16, device_map="auto", cache_dir='/scratch/gpfs/ij9216/cached_models')

    print("TESTING AA1")
    tokenizer = AutoTokenizer.from_pretrained(model_id, padding_side = "left")
    tokenizer.pad_token_id = tokenizer.eos_token_id

    print("TESTING AA2")
    #print model size
    module_sizes = accelerate.utils.compute_module_sizes(model)
```

```
    print(f"The model size is {module_sizes[''] * 1e-9} GB")

    if torch.cuda.is_available():
        gpu_id = 0  # or whichever GPU you're using
        total_memory = torch.cuda.get_device_properties(gpu_id).total_memory
        print(f"Total GPU memory: {total_memory / 1024**3:.2f} GB")

    with open(prompt_path, 'r') as file:
        promptsDict = json.load(file)

    with open(data_path, 'r') as file:
        data = json.load(file)
    #get the first X datapoints
    numSamplesTotal = len(data["references"])

    references=data["references"][0:numSamplesTotal]
    labels=data["labels"][0:numSamplesTotal]
    print("dataset " + str(data_path) + " loaded")



    subtexts=None
    if prompting_type== "single":
        prompts=promptsDict["prompt_singleStep"]
        subtexts=None
    elif prompting_type== "multi_parallel":
        if not random_subtext:
            prompts=promptsDict["prompts_subtext_multiStep"]
            subtexts=promptsDict["subtexts"]
        else:
            prompts=promptsDict["prompts_subtext_multiStep_random"]
            subtexts = promptsDict["randomSubtextList"]
```

```
    elif prompting_type== "voting_choice":
        prompts = promptsDict["prompts_subtext_voting1"]
        subtexts = promptsDict["subtexts"]
    elif prompting_type== "voting_score":
        prompts = promptsDict["prompts_subtext_voting2"]
        subtexts=promptsDict["subtexts"]
    elif prompting_type== "voting_choice_rigged":
        prompts = promptsDict["prompts_subtext_voting3"]
        subtexts=promptsDict["subtexts"]
    elif prompting_type== "voting_score_rigged":
        prompts = promptsDict["prompts_subtext_voting4"]
        subtexts=promptsDict["subtexts"]
    if random_thought:
        random_thought_list = pd.read_csv("data/
RandomGens20250409_Llama8B.csv")["GeneratedText"].tolist()
    else:
        random_thought_list = None

    print(f"Using model: {model_name}")
    print(f"Data path: {data_path}")
    print(f"Prompt path: {prompt_path}")
    print(f"Prompting type: {prompting_type}")

    all_res_dfs = []

    for shuffle_id in range(1, num_shuffles + 1):
        original_indices = list(range(len(prompts[0])))
        shuffled_indices = original_indices[:]
        if random_shuffle:
            if shuffle_id != 1:  # don't shuffle the first time
                random.shuffle(shuffled_indices)
            if prompting_type == "single":
```

```
            pass
        elif prompting_type == "multi_parallel":
            prompts[0] = [prompts[0][i] for i in shuffled_indices]
        elif prompting_type == "voting_score":
            prompts[0] = [prompts[0][i] for i in shuffled_indices]
        elif prompting_type == "voting_choice":
            prompts[0] = [prompts[0][i] for i in shuffled_indices]
        elif prompting_type == "voting_choice_rigged":
            prompts[0] = [prompts[0][i] for i in shuffled_indices]

        if subtexts is not None:
            subtexts = [subtexts[i] for i in shuffled_indices]

    print(prompts)

    today_date = datetime.now().strftime("%Y%m%d")
    start_time = datetime.now()

    results_dict = performAnalysisOnWholeData(references,labels, prompts, prompting_type,model_name, model, tokenizer, bertscore, shuffled_indices,subtexts,_temperature=_temperature, random_thought_list=random_thought_list)

    res_df=pd.DataFrame.from_dict(results_dict)


    #almost exactly the same, just need to get reference/label from csv instead of json file like before
```

```
        #and then also add another column which says if voted subtext index matches real subtext index, (i.e. whether generation was correct

        #also want to do it with multi parallel realistically

        #can probably handle almost everything within the performAnalysisOnWholeData function, only the input needs to change asw

        res_df.insert(0, 'sample_id', range(1, numSamplesTotal + 1))
        res_df.insert(1, 'shuffle_id', shuffle_id)

        res_df['shuffle_indices'] = [shuffled_indices] * numSamplesTotal
        res_df['subtexts'] = [subtexts] * numSamplesTotal

        all_res_dfs.append(res_df)

    final_res_df = pd.concat(all_res_dfs, ignore_index=True)
    print(final_res_df.head())

    # Calculate the duration
    duration = datetime.now() - start_time
    print(f"Time taken: {duration}")

    # save the df to a csv
    # filename: file_path _ prompt_file _ prompt index _ model name _ date
    res_f_name = f"noramlrand{today_date}_{data_path.stem}_{prompt_path.stem}_{model_name}_{prompting_type}_shuffle{num_shuffles}_temp{_temperature}_random{str(random_subtext)}
```

```
_randomThought{str(random_thought)}.csv"

    # Check and create the directory
    if not os.path.exists("results"):
        os.makedirs("results")
    final_res_df.to_csv(f"results/{res_f_name}")

    print(f"Results saved to results/{res_f_name}")


if __name__ == "__main__":
    main()
```

```

    import torch
import time, subprocess, argparse, json, re, math
import accelerate.utils
from transformers import AutoModelForCausalLM, AutoTokenizer,
    AutoModel
from evaluate import load

#default values from generation_config.json   "temperature": 0.6, "
    top_p": 0.9,

def generateInference(inputPrompts, model, tokenizer, modelType="
    Llama8B",debugTime=False, device='cuda',_temperature=0.6):

    #tokenizing inputs
    texts = tokenizer.apply_chat_template(inputPrompts,
    add_generation_prompt=True, tokenize=False)
    inputs = tokenizer(texts, padding=True, return_tensors="pt").to(
    device)

```

```
 temp_texts=tokenizer.batch_decode(inputs["input_ids"], skip_special_tokens=True)#to see how tokenizer structured input

 if debugTime:
     #generate inferred next sentence
     print("Starting generation")
     start_time = time.time()

 if (modelType=="Llama1B"):
     terminators = [
         tokenizer.eos_token_id,
         tokenizer.convert_tokens_to_ids("<|eot_id|>")
     ]
     if _temperature==0.0:
         gen_tokens = model.generate(
             **inputs,
             max_new_tokens=50,
             pad_token_id=tokenizer.eos_token_id,eos_token_id=terminators,
             do_sample=False,
         )
     else:
         gen_tokens = model.generate(
             **inputs,
             max_new_tokens=50,
             pad_token_id=tokenizer.eos_token_id,eos_token_id=terminators,
             temperature=_temperature,
         )
 else:
     if _temperature==0.0:
         gen_tokens = model.generate(
```

```
                **inputs,
                max_new_tokens=50,
                pad_token_id=tokenizer.eos_token_id,
                do_sample=False,
            )
        else:
            gen_tokens = model.generate(
                **inputs,
                max_new_tokens=50,
                pad_token_id=tokenizer.eos_token_id,
                temperature=_temperature,
            )

    if debugTime:
        print(f"Generation finished, Time: {time.time()-start_time}")

    #decode generated text and remove the prompt
    gen_text = tokenizer.batch_decode(gen_tokens, skip_special_tokens=True)
    gen_text = [i[len(temp_texts[idx]):] for idx, i in enumerate(gen_text)]
    return gen_text
```

```
import torch
import time, subprocess, argparse, json, re, math
import accelerate.utils
import numpy as np
import pandas as pd
from transformers import AutoModelForCausalLM, AutoTokenizer, AutoModel
from evaluate import load
from pathlib import Path
```

```python
from datetime import datetime
import os
import math
import random
from inference import generateInference
from distutils.util import strtobool



def generateRandomText(_references, sys_prompt, model, tokenizer, modelType="Llama8B", bertscore_model="distilbert-base-uncased", _temperature=1.0):
    myinput=[]
    for i in range (0,len(_references)):
        myinput.append([{"role": "system", "content": sys_prompt}, {"role": "user", "content": _references[i]}] )
    #getting next sentence and outputting similarity results
    gen_text=generateInference(myinput, model, tokenizer, modelType, _temperature=_temperature)
    print("Generated text: with tempeature " + str(_temperature))

    return gen_text


def main():
    parser = argparse.ArgumentParser(description="Run analysis with specified model and data.")
    parser.add_argument(’--model_name’, type=str, required=True, help=’Name of the LM to use’)
    args = parser.parse_args()

    model_name = args.model_name
```

```python
    #set up device to GPU if available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")
    #LOADING AND SETTING UP THE MODEL
    #set up model id based on model type

    if (model_name=="Llama1B"):
        model_id = "meta-llama/Llama-3.2-1B-Instruct"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.bfloat16, device_map="auto").to(device)
    elif (model_name=="Llama8B"):
        model_id = "meta-llama/Llama-3.1-8B-Instruct"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.bfloat16, device_map="auto").to(device)
    elif (model_name=="Llama70B"):
        model_id ="hugging-quants/Meta-Llama-3.1-70B-Instruct-GPTQ-INT4"
        #model_id = "unsloth/Meta-Llama-3.1-70B-Instruct-bnb-4bit"
        model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16).to(device)
        # model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto", cache_dir='/scratch/gpfs/ij9216/cached_models').to(device)
        # model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16, device_map="auto", cache_dir='/scratch/gpfs/ij9216/cached_models')

    print("TESTING AA1")
    tokenizer = AutoTokenizer.from_pretrained(model_id, padding_side = "left")
    tokenizer.pad_token_id = tokenizer.eos_token_id
```

```
    print("TESTING AA2")
    #print model size
    module_sizes = accelerate.utils.compute_module_sizes(model)
    print(f"The model size is {module_sizes[''] * 1e-9} GB")

    if torch.cuda.is_available():
        gpu_id = 0  # or whichever GPU you're using
        total_memory = torch.cuda.get_device_properties(gpu_id).total_memory
        print(f"Total GPU memory: {total_memory / 1024**3:.2f} GB")


    numGens=8000
    batchSize=50
    references=["Generate some random text" for _ in range(numGens)]

    numBatches=math.ceil(len(references)/batchSize)

    allGeneratedText=[]
    for i in range(numBatches):
        startIndex=i*batchSize
        endIndex=min((i+1)*batchSize,len(references))
        batchReferences=references[startIndex:endIndex]
        #generate random text
        generatedText=generateRandomText(batchReferences, "Generate some random text", model, tokenizer, modelType="Llama8B", bertscore_model="distilbert-base-uncased",_temperature=2.0)
        allGeneratedText.extend(generatedText)

    results_dict = {"GeneratedText": allGeneratedText}
    final_res_df = pd.DataFrame.from_dict(results_dict)
```

```python


    today_date = datetime.now().strftime("%Y%m%d")


    # save the df to a csv
    # filename: file_path _ prompt_file _ prompt index _ model name _ date
    res_f_name = f"RandomGens{today_date}_{model_name}.csv"

    # Check and create the directory
    if not os.path.exists("results"):
        os.makedirs("results")
    final_res_df.to_csv(f"results/{res_f_name}")

    print(f"Results saved to results/{res_f_name}")


if __name__ == "__main__":
    main()
```

A.5 Language model hyperparameters

Hyperparameters reterived from ”generation$_c onfig.json”.ItisthesameforLlaMa3.11B, LlaMa3.28$

```json
{
  "bos_token_id": 128000,
  "do_sample": true,
  "eos_token_id": [
    128001,
    128008,
    128009
  ],
```

```
  "temperature": 0.6,
  "top_p": 0.9,
  "transformers_version": "4.46.2"
}
```

A.6 Banded Ridge Regression Code

dataframe_setup.py

```
import pandas as pd


#load transcript and extract sentences
# Extract sentences/sentence indices and add them to the transcript
    DataFrame.
puncTranscript_path = "monkey_util-whisperx_transcript.csv" # Change
     as needed

df = pd.read_csv(puncTranscript_path)
df.dropna(subset=['start'], inplace=True)
df.sort_values("start", inplace=True)
df.word.tail(10)

df.index = list(range(len(df.word)))# fix word index numbering
sentenceEnds = df.index[df.word.str.contains('[.?!]')]
df.loc[sentenceEnds]
sentenceBegs = sentenceEnds + 1
sentenceBegs = sentenceBegs.insert(0, 0)
sentenceBegs = sentenceBegs.delete(-1)

sentenceIdx = [0] * len(df.word);
sentence = [''] * len(df.word);
```

```
for i, j, k in zip(sentenceBegs, sentenceEnds, range(len(sentenceBegs))):
  fullSentence = (df.word[i:(j+1)]).str.cat(sep=' ')#concatenate words in sentence

  sentenceIdx[i:(j+1)] = [k] * len(range(i, (j+1)))#set sentence index for each word in sentence to be the same
  sentence[i:(j+1)] = [fullSentence] * len(range(i, (j+1)))

sentenceIdx = pd.DataFrame({'sentence_idx': sentenceIdx})
sentence = pd.DataFrame({'sentence': sentence})
df = df.join(sentenceIdx)
df = df.join(sentence)
df.insert(0, "word_idx", df.index.values)

df.to_csv('transcript_inc_sentence.csv', index=False)
```

create_embeddings.py

```
import h5py
import torch
import numpy as np
import pandas as pd

from himalaya.backend import set_backend, get_backend

from accelerate import Accelerator, find_executable_batch_size
from transformers import AutoModelForCausalLM, AutoTokenizer
from sentence_transformers import SentenceTransformer
import gensim.downloader
import re

if torch.cuda.is_available():
    set_backend("torch_cuda")
```

```
    print("Using cuda!")


#load transcript and extract sentences
# Extract sentences/sentence indices and add them to the transcript DataFrame.
transcript_path = "transcript_inc_sentence.csv" # Change as needed
df = pd.read_csv(transcript_path)

# Download 300-dimensional word2vec embeddings
model_name = 'word2vec-google-news-300'
n_features = 300

model = gensim.downloader.load(model_name)


#creating static word embeddings
transcript_w2v = df.copy()

# Convert words to lowercase
transcript_w2v['word'] = transcript_w2v.word.str.lower()
# Filter words
transcript_w2v['filtered_word'] = transcript_w2v.word.apply(lambda word: re.sub(r"[^\w\s]", "", word))

# Function to extract embeddings if available
def get_vector(word):
    #word = ''.join(char for char in word if char.isalnum() and char not in ["'", ",", ".","?","!","-"])
    if word in model.key_to_index:
        return model.get_vector(word, norm=True).astype(np.float32)
    return np.nan
```

```
# Extract embedding for each word
transcript_w2v['embedding'] = transcript_w2v.filtered_word.apply(
    get_vector)  # word2vec embeddings are 300-dimensional
# Insert zero vectors for entries with no embeddings
embedding_length = n_features  # Length of the embeddings (300 for
    word2vec)
transcript_w2v['embedding'] = transcript_w2v['embedding'].apply(
    lambda x: np.zeros(embedding_length, dtype=np.float32) if np.any
    (np.isnan(x)) else x
)


# Print out words not found in vocabulary
print(f'{(transcript_w2v.embedding.isna()).sum()} words not found:')
print(transcript_w2v.filtered_word[transcript_w2v.embedding.isna()].
    value_counts())


#creating contextual word embeddings
#model to tokenize the data
modelname = "gpt2"
context_len = 32
device = torch.device("cpu")
if torch.cuda.is_available():
    device = torch.device("cuda", 0)
    print("Using cuda!")

model_sentence = SentenceTransformer("intfloat/e5-mistral-7b-
    instruct")
# In case you want to reduce the maximum sequence length:
model_sentence.max_seq_length = 1000
```

```python

# Load model for tokenization
tokenizer = AutoTokenizer.from_pretrained(modelname)

#explode dataframe using tokens
df["hftoken"] = df.word.apply(lambda x: tokenizer.tokenize(" " + x))

df = df.explode("hftoken", ignore_index=True)
df["token_id"] = df.hftoken.apply(tokenizer.convert_tokens_to_ids)

df.head(10)

#model to get token embeddings
print("Loading model...")
model = AutoModelForCausalLM.from_pretrained(modelname)

print(
    f"Model : {modelname}"
    f"\nLayers: {model.config.num_hidden_layers}"
    f"\nEmbDim: {model.config.hidden_size}"
    f"\nConfig: {model.config}"
)
model = model.eval()
model = model.to(device)

# setting up data for getting embeddings (using previous 32 tokens
    as context)
# could change this to use e.g. last sentence as context
token_ids = df.token_id.tolist()

print(f"Token id length: {len(token_ids)}")

```

```python
fill_value = 0
if tokenizer.pad_token_id is not None:
    fill_value = tokenizer.pad_token_id

#data ends up being an array of length token_ids containing arrays
    of length context_len + 1
data = torch.full((len(token_ids), context_len + 1), fill_value,
    dtype=torch.long)
for i in range(len(token_ids)):
    example_tokens = token_ids[max(0, i - context_len) : i + 1]
    data[i, -len(example_tokens) :] = torch.tensor(example_tokens)


print(f"Data has a shape of: {data.shape}") #  torch.Size([5749,
    33])


#extract embeddings of all of the tokens
#use accekerator to make extracting features more efficient
accelerator = Accelerator()
@find_executable_batch_size(starting_batch_size=32)
def inference_loop(batch_size=32):
    # nonlocal accelerator  # Ensure they can be used in our context
    accelerator.free_memory()  # Free all lingering references

    data_dl = torch.utils.data.DataLoader(
        data, batch_size=batch_size, shuffle=False
        )

    top_guesses = []
    ranks = []
    true_probs = []
```

```
    entropies = []
    embeddings = []

    #batch size is number of entries in each batch (i.e 32,32... (something less than 32 at end))

    with torch.no_grad():
        i=0
        for batch in data_dl:
            # Get output from model
            output = model(batch.to(device), output_hidden_states=True)
            logits = output.logits
            states = output.hidden_states

            true_ids = batch[:, -1]
            brange = list(range(len(true_ids)))
            logits_order = logits[:, -2, :].argsort(descending=True)
            batch_top_guesses = logits_order[:, 0]
            batch_ranks = torch.eq(logits_order, true_ids.reshape(-1, 1).to(device)).nonzero()[:, 1]
            batch_probs = torch.softmax(logits[:, -2, :], dim=-1)
            batch_true_probs = batch_probs[brange, true_ids]
            batch_entropy = torch.distributions.Categorical(probs=batch_probs).entropy()

            #hidden states is a list of tensors where each tensor has shape (batch_size, sequence length, hidden_size)
            #hidden size is dimensitonality of the model's hidden representations

            #extracts the hidden state for the last token of each
```

```
sequence in the batch, giving a list of 13 tensors of shape (batch size, hidden size) i.e (32, 768)
            batch_embeddings = [state[:, -1, :].numpy(force=True) for state in states ]


            top_guesses.append(batch_top_guesses.numpy(force=True))
            ranks.append(batch_ranks.numpy(force=True))
            true_probs.append(batch_true_probs.numpy(force=True))
            entropies.append(batch_entropy.numpy(force=True))
            embeddings.append(batch_embeddings)
            i+=1
        print(f"Number of batches is {i}")
        return top_guesses, ranks, true_probs, entropies, embeddings

top_guesses, ranks, true_probs, entropies, embeddings = inference_loop()

#embeddings is a list of 180 lists, each list has 13 tensors of shape (32, 768)
#GPT 2 has 13 layers of embeddings, each word embedding is 768 dimensions long
print(f"There are {len(embeddings[0])} layers of embeddings")
print(f"Each word embedding is {embeddings[0][0].shape[1]} dimensions long")


embeddingsA=np.hstack(embeddings)#this should be len 180, containing a bunch of ndarrays of shape (32,768)
print(embeddingsA.shape)
embeddingsB=embeddingsA[12,:,:]
print(embeddingsB.shape)# Shape: (num tokens, embedding size)
```

```
print("List shape")
print(len(embeddingsB.tolist()))
print(len(embeddingsB.tolist()[0]))

df["rank"] = np.concatenate(ranks)
df["true_prob"] = np.concatenate(true_probs)
df["top_pred"] = np.concatenate(top_guesses)
df["entropy"] = np.concatenate(entropies)
df["tkn_embedding"] = embeddingsB.tolist()
df.head(10)


#average embeddings over tokens in each word, and add to dataframe
aligned_embeddings = []
for _, group in df.groupby("word_idx"): # group by word index
    indices = group.index.to_numpy()
    average_emb = embeddingsB[indices].mean(0) # average features
    aligned_embeddings.append(average_emb)
aligned_embeddings = np.stack(aligned_embeddings)
print(f"LLM embeddings matrix has shape: {aligned_embeddings.shape}"
    )

print(len(aligned_embeddings.tolist()))

#get list of sentences in order
# Get the indices of the first occurrence of each unique sentence
unique_indices = df["sentence_idx"].drop_duplicates(keep="first").
    index.tolist()
# Access each sentence in order
sentences = df.loc[unique_indices, "sentence"].tolist()

```

```
#create new list of embeddings for each sentence
sentence_embeddings=[]
sentence_embeddings = model_sentence.encode(sentences)
dataLength=df.shape[0]

# explode the sentence list to line up with words(for each word add its sentence embedding to array)
unique_word_indices = df["word_idx"].drop_duplicates(keep="first").index.tolist()
expanded_sentence_embeddings=[]
for idx in unique_word_indices:
  curSentence=df["sentence_idx"].iloc[idx]
  expanded_sentence_embeddings.append(sentence_embeddings[curSentence])

print(len(expanded_sentence_embeddings))


df_sentence_embeddings = pd.DataFrame({'sentence_embedding':expanded_sentence_embeddings})
df_word_embeddings = pd.DataFrame({'word_embedding':aligned_embeddings.tolist()})
df_static_word_embeddings =pd.DataFrame({'static_word_embedding':transcript_w2v.embedding.tolist()})

df_word_embeddings = df_word_embeddings.join(df_sentence_embeddings)
df_word_embeddings = df_word_embeddings.join(df_static_word_embeddings)



print(df_word_embeddings.head())
```

```
df_word_embeddings.to_hdf('embeddings.h5', key='df', mode='w')
```

joint_encoding.py

```
import mne
import h5py
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from nilearn.plotting import plot_markers

from himalaya.backend import set_backend, get_backend
from himalaya.ridge import RidgeCV
from himalaya.scoring import correlation_score
from himalaya.scoring import correlation_score_split

from sklearn.model_selection import KFold
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

from himalaya.kernel_ridge import Kernelizer, ColumnKernelizer,
    MultipleKernelRidgeCV
import joblib

if torch.cuda.is_available():
    set_backend("torch_cuda")
    print("Using cuda!")

#span of the epochs
tMin=-6.0
tMax=6.0

#sampling rate
```

```python
sampling_rate = 512
step_size=16

#load transcript and load embeddings
transcript_path = "transcript_inc_sentence.csv"
df = pd.read_csv(transcript_path)

df_embeddings= pd.read_hdf('embeddings.h5', 'df')
#print(df_embeddings.head())
aligned_embeddings = np.vstack(df_embeddings["word_embedding"].values)
sentence_embeddings = np.vstack(df_embeddings["sentence_embedding"].values)
static_word_embeddings = np.vstack(df_embeddings["static_word_embedding"].values)


print(aligned_embeddings.shape)
print(sentence_embeddings.shape)
print(static_word_embeddings.shape)

#construct separate dataframe containins words with their start and end times
df_word = df.groupby("word_idx").agg(dict(word="first", start="first", end="last"))

#load the preprocessed brain data
file_path = "sub-03_task-podcast_desc-highgamma_ieeg.fif"
raw = mne.io.read_raw_fif(file_path, verbose=False)
picks = mne.pick_channels_regexp(raw.ch_names, "LG[AB]*")
raw = raw.pick(picks)

```

```python
print(raw)

#map seconds to samples by multiplying to sampling rate
events = np.zeros((len(df_word), 3), dtype=int)
events[:, 0] = (df_word.start * raw.info['sfreq']).astype(int)
print(events.shape)

#create epochs and downsample to 32 Hz
epochs = mne.Epochs(raw,events,tmin=tMin,tmax=tMax,baseline=None,proj=False,event_id=None,preload=True,event_repeated="merge")
print(f"Epochs object has a shape of: {epochs._data.shape}")
epochs = epochs.resample(sfreq=32, npad='auto', method='fft', window='hamming')
print(f"Epochs object has a shape of: {epochs._data.shape}")


#create Y matrices
#reshape target matrix Y by horizontally stacking electodes and lags along the second dimension
epochs_data = epochs.get_data(copy=True)
epochs_data = epochs_data.reshape(len(epochs), -1)
print(f"ECoG data matrix shape: {epochs_data.shape}")

#align features with target matrix/ECoG data (some bad epochs may be dropped)
selected_df = df_word.iloc[epochs.selection]
averaged_embeddings = aligned_embeddings[epochs.selection]
averaged_sentence_embeddings=sentence_embeddings[epochs.selection]


#Change the float precision to float32 for all data to take advantage of the GPU memory and computational speed.
```

```
X_word_ctx = averaged_embeddings
X_word_static = static_word_embeddings[epochs.selection]
X_sentence=averaged_sentence_embeddings
X_word_used=X_word_ctx

X_joint = np.hstack([X_word_used, X_sentence]) # Horizontal-stack both embeddings to create joint model
Y = epochs_data

print(f"Joint predictor matrix shape: {X_joint.shape}")
print(f"X_word_ctx shape: {X_word_ctx.shape}")
print(f"X_word_static shape: {X_word_static.shape}")
print(f"X_sentence shape: {X_sentence.shape}")
print(f"Y shape: {Y.shape}")

#change float precision to float32 for all data to take advantage of GPU memory/computational speed
if "torch" in get_backend().__name__:
    X_joint = X_joint.astype(np.float32)
    X_word_ctx=X_word_ctx.astype(np.float32)
    X_word_static=X_word_static.astype(np.float32)
    Y = Y.astype(np.float32)

#do the banded ridge regression

# Cross validation is done with an outer part split into 2 and inner part split into 5
inner_cv = KFold(n_splits=5)

# Make pipeline with kernelizer for each feature space
column_pipeline = make_pipeline(
    StandardScaler(with_mean=True, with_std=True),
```

```
    Kernelizer(kernel="linear"),
)

#set up slices
width_w = X_word_used.shape[1]
width_s = X_sentence.shape[1]

slice_w = slice(0, width_w)
slice_s = slice(width_w, width_w + width_s)
print(f"Word slice: {slice_w}")
print(f"Sentence slice: {slice_s}")

# Compile joint column kernelizer
column_kernelizer = ColumnKernelizer(
    [('word', column_pipeline, slice_w),
     ('sentence', column_pipeline, slice_s)])


# Ridge regression with alpha grid and nested CV
solver = 'random_search'
n_iter = 20
alphas = np.logspace(1, 10, 10)
solver_params = dict(n_iter=n_iter, alphas=alphas)

# Banded ridge regression with column kernelizer
banded_ridge = MultipleKernelRidgeCV(kernels="precomputed", solver=solver, solver_params=solver_params, cv=inner_cv)

# Chain transfroms and estimator into pipeline
pipeline = make_pipeline(column_kernelizer, banded_ridge)

def train_joint_encoding(X, Y,_split=False,_epochs_shape=None):
```

```
    
    corrs = [] # empty array to store correlation results
    kfold = KFold(2, shuffle=False) # outer 2-fold cross-validation setup
    for train_index, test_index in kfold.split(X): # loop through folds

        # Split train and test datasets
        X1_train, X1_test = X[train_index], X[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]

        # Standardize Y
        scaler = StandardScaler()
        Y_train = scaler.fit_transform(Y_train)
        Y_test = scaler.transform(Y_test)

        pipeline.fit(X1_train, Y_train) # Fit pipeline with transforms and ridge estimator

        if _split:
            Y_preds = pipeline.predict(X1_test, split=True) # Use trained model to predict on test set
            corr = correlation_score_split(Y_test, Y_preds) # Compute correlation score
        else:
            Y_preds = pipeline.predict(X1_test)
            corr = correlation_score(Y_test, Y_preds).reshape(epochs_shape) # Compute correlation score

        if "torch" in get_backend().__name__: # if using gpu, transform tensor back to numpy
            corr = corr.numpy(force=True)
```

```python

        corrs.append(corr) # append fold correlation results to final results
    return np.stack(corrs)


alphas = np.logspace(1, 10, 10) # specify alpha values
inner_cv_single = KFold(n_splits=5, shuffle=False) # inner 5-fold cross-validation setup
single_model = make_pipeline(
    StandardScaler(), RidgeCV(alphas, fit_intercept=True, cv=inner_cv_single) # pipeline
)


def train_single_encoding(X, Y,_epochs_shape):

    corrs = [] # empty array to store correlation results
    kfold = KFold(2, shuffle=False) # outer 2-fold cross-validation setup
    for train_index, test_index in kfold.split(X): # loop through folds

        # Split train and test datasets
        X1_train, X1_test = X[train_index], X[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]

        # Standardize Y
        scaler = StandardScaler()
        Y_train = scaler.fit_transform(Y_train)
        Y_test = scaler.transform(Y_test)

```

```
        single_model.fit(X1_train, Y_train) # Fit pipeline with transforms and ridge estimator
        Y_preds = single_model.predict(X1_test) # Use trained model to predict on test set
        corr = correlation_score(Y_test, Y_preds).reshape(_epochs_shape) # Compute correlation score

        if "torch" in get_backend().__name__: # if using gpu, transform tensor back to numpy
            corr = corr.numpy(force=True)

        corrs.append(corr) # append fold correlation results to final results
    return np.stack(corrs)


epochs_shape = epochs._data.shape[1:] # number of electrodes * number of lags
corrs_embedding = train_joint_encoding(X_joint, Y, _split=False, _epochs_shape=epochs_shape)
print(f"Encoding performance correlating matrix shape: {corrs_embedding.shape}")
# Encoding performance correlating matrix shape: (2, 127, 128)

corrs_embedding_split = train_joint_encoding(X_joint, Y, True)
corrs_embedding_split = corrs_embedding_split.reshape(2, 2, *epochs_shape)
print(f"Encoding performance correlating matrix shape: {corrs_embedding_split.shape}")
#Encoding performance correlating matrix shape: (2, 2, 127, 128)
corrs_embedding_word_ctx = train_single_encoding(X_word_ctx, Y, _epochs_shape=epochs_shape)  # predictions with just contextual
```

```python
embeddings
print(f"Encoding performance correlating matrix shape: {corrs_embedding_word_ctx.shape}")
# Encoding performance correlating matrix shape: (2, 127, 128)
corrs_embedding_word_static = train_single_encoding(X_word_static, Y, _epochs_shape=epochs_shape)  # predictions with just contextual embeddings
print(f"Encoding performance correlating matrix shape: {corrs_embedding_word_ctx.shape}")

#127 electrodes, 128 lags

joblib.dump(corrs_embedding, 'corrs_embedding_joint_ctx_sentence.pkl')
joblib.dump(corrs_embedding_split, 'corrs_embedding_split_ctx_sentence.pkl')
joblib.dump(corrs_embedding_word_ctx, 'corrs_embedding_word_ctx.pkl')
joblib.dump(corrs_embedding_word_static, 'corrs_embedding_word_static.pkl')

#Note:
#    - The dimensions of the following arrays are:
#        - corrs_embedding: (2, number_of_electrodes, number_of_lags)
#        - corrs_embedding_split: (2, 2, number_of_electrodes, number_of_lags)
#        - corrs_embedding_word_ctx: (2, number_of_electrodes, number_of_lags)


print("Done!")
```

Appendix B

Engineering Standards

This independent project adheres to the following engineering and industrial standards:

- **Programming Languages**: Python,
- **Open Source Software**: NumPy, PyTorch, HuggingFace Transformers Library

Standards Compliance

All tools, programming languages, and methodologies employed in the development of this thesis conform to well-established academic and industrial standards. This adherence ensures that both the theoretical insights and the associated codebases are robust, scalable, and easily adaptable to a variety of practical applications in the engineering domain. Additionally, the thesis consistently utilizes the International System of Units (SI units), ensuring standardization and precision in the presentation of technical data.

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," June 2017.

[2] F. Xu, Q. Hao, Z. Zong, J. Wang, Y. Zhang, J. Wang, X. Lan, J. Gong, T. Ouyang, F. Meng, C. Shao, Y. Yan, Q. Yang, Y. Song, S. Ren, X. Hu, Y. Li, J. Feng, C. Gao, and Y. Li, "Towards large reasoning models: A survey of reinforced reasoning with large language models," 2025.

[3] A. Goldstein, Z. Zada, E. Buchnik, M. Schain, A. Price, B. Aubrey, S. A. Nastase, A. Feder, D. Emanuel, A. Cohen, A. Jansen, H. Gazula, G. Choe, A. Rao, C. Kim, C. Casto, L. Fanda, W. Doyle, D. Friedman, P. Dugan, L. Melloni, R. Reichart, S. Devore, A. Flinker, L. Hasenfratz, O. Levy, A. Hassidim, M. Brenner, Y. Matias, K. A. Norman, O. Devinsky, and U. Hasson, "Shared computational principles for language processing in humans and deep language models," *Nat. Neurosci.*, vol. 25, pp. 369–380, Mar. 2022.

[4] Z. Wu, L. Qiu, A. Ross, E. Akyürek, B. Chen, B. Wang, N. Kim, J. Andreas, and Y. Kim, "Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks," 2023.

[5] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro,

and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4," 2023.

[6] J. Gauthier, J. Hu, E. Wilcox, P. Qian, and R. Levy, "SyntaxGym: An online platform for targeted evaluation of language models," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2020.

[7] A. Warstadt and S. R. Bowman, "What artificial neural networks can tell us about human language acquisition," 2022.

[8] K. Mahowald, A. A. Ivanova, I. A. Blank, N. Kanwisher, J. B. Tenenbaum, and E. Fedorenko, "Dissociating language and thought in large language models," 2023.

[9] A. M. Paunov, I. A. Blank, O. Jouravlev, Z. Mineroff, J. Gallée, and E. Fedorenko, "Differential tracking of linguistic vs. mental state content in naturalistic stimuli by language and theory of mind (ToM) brain networks," *Neurobiol. Lang. (Camb.)*, vol. 3, pp. 413–440, July 2022.

[10] E. Fedorenko, A. A. Ivanova, and T. I. Regev, "The language network as a natural kind within the broader landscape of the human brain," *Nat. Rev. Neurosci.*, vol. 25, pp. 289–312, May 2024.

[11] E. Fedorenko and R. Varley, "Language and thought are not the same thing: evidence from neuroimaging and neurological patients," *Ann. N. Y. Acad. Sci.*, vol. 1369, pp. 132–153, Apr. 2016.

[12] G. Granato, A. M. Borghi, and G. Baldassarre, "A computational model of language functions in flexible goal-directed behaviour," *Sci. Rep.*, vol. 10, p. 21623, Dec. 2020.

[13] T. Ruffman, L. Slade, and E. Crowe, "The relation between children's and mothers' mental state language and theory-of-mind understanding," *Child Dev.*, vol. 73, pp. 734–751, May 2002.

[14] O. et al., "OpenAI o1 system card," Dec. 2024.

[15] D. et al., "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," 2025.

[16] Q. Team, "Qwq: Reflect deeply on the boundaries of the unknown," Nov 2024.

[17] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2022.

[18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are Zero-Shot reasoners," 2022.

[19] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving factuality and reasoning in language models through multiagent debate," 2023.

[20] S. Chen, B. Li, and D. Niu, "Boosting of thoughts: Trial-and-error problem solving with large language models," 2024.

[21] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," 2023.

[22] L. Yang, Z. Yu, T. Zhang, S. Cao, M. Xu, W. Zhang, J. E. Gonzalez, and B. Cui, "Buffer of thoughts: Thought-augmented reasoning with large language models," 2024.

[23] S. Parashar, B. Olson, S. Khurana, E. Li, H. Ling, J. Caverlee, and S. Ji, "Inference-time computations for LLM reasoning and planning: A benchmark and insights," 2025.

[24] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023.

[25] L. Chen, J. Q. Davis, B. Hanin, P. Bailis, I. Stoica, M. Zaharia, and J. Zou, "Are more LLM calls all you need? towards scaling laws of compound inference systems," 2024.

[26] M. K. Ho, R. Saxe, and F. Cushman, "Planning with theory of mind," *Trends Cogn. Sci.*, vol. 26, pp. 959–971, Nov. 2022.

[27] T. Ullman, "Large language models fail on trivial alterations to Theory-of-Mind tasks," 2023.

[28] N. Shapira, M. Levy, S. H. Alavi, X. Zhou, Y. Choi, Y. Goldberg, M. Sap, and V. Shwartz, "Clever hans or neural theory of mind? stress testing social reasoning in large language models," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)* (Y. Graham and M. Purver, eds.), (St. Julian's, Malta), pp. 2257–2273, Association for Computational Linguistics, Mar. 2024.

[29] H. Kim, M. Sclar, X. Zhou, R. Bras, G. Kim, Y. Choi, and M. Sap, "FANToM: A benchmark for stress-testing machine theory of mind in interactions," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (H. Bouamor, J. Pino, and K. Bali, eds.), (Singapore), pp. 14397–14413, Association for Computational Linguistics, Dec. 2023.

[30] K. Gandhi, J.-P. Fränken, T. Gerstenberg, and N. Goodman, "Understanding social reasoning in language models with language models," in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[31] C. Jin, Y. Wu, J. Cao, J. Xiang, Y.-L. Kuo, Z. Hu, T. Ullman, A. Torralba, J. Tenenbaum, and T. Shu, “MMToM-QA: Multimodal theory of mind question answering,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), (Bangkok, Thailand), pp. 16077–16102, Association for Computational Linguistics, Aug. 2024.

[32] Y. Wu, Y. He, Y. Jia, R. Mihalcea, Y. Chen, and N. Deng, “Hi-ToM: A benchmark for evaluating higher-order theory of mind reasoning in large language models,” in *Findings of the Association for Computational Linguistics: EMNLP 2023* (H. Bouamor, J. Pino, and K. Bali, eds.), (Singapore), pp. 10691–10706, Association for Computational Linguistics, Dec. 2023.

[33] M. Sclar, S. Kumar, P. West, A. Suhr, Y. Choi, and Y. Tsvetkov, “Minding language models’ (lack of) theory of mind: A plug-and-play multi-character belief tracker,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), (Toronto, Canada), pp. 13960–13980, Association for Computational Linguistics, July 2023.

[34] Y. Gu, O. Tafjord, H. Kim, J. Moore, R. L. Bras, P. Clark, and Y. Choi, “SimpleToM: Exposing the gap between explicit ToM inference and implicit ToM application in LLMs,” 2024.

[35] C. Jung, D. Kim, J. Jin, J. Kim, Y. Seonwoo, Y. Choi, A. Oh, and H. Kim, “Perceptions to beliefs: Exploring precursory inferences for theory of mind in large language models,” 2024.

[36] J. Wu, Z. Chen, J. Deng, S. Sabour, H. Meng, and M. Huang, “COKE: A cognitive knowledge graph for machine theory of mind,” May 2023.

[37] H. Kim, M. Sclar, T. Zhi-Xuan, L. Ying, S. Levine, Y. Liu, J. B. Tenenbaum, and Y. Choi, "Hypothesis-driven theory-of-mind reasoning for large language models," Feb. 2025.

[38] A. Afrasiyabi, E. Busch, R. Singh, D. Bhaskar, L. Caplette, N. Turk-Browne, and S. Krishnaswamy, "Looking through the mind's eye via multimodal encoder-decoder networks," 2024.

[39] G. Tuckute, N. Kanwisher, and E. Fedorenko, "Language in brains, minds, and machines," *Annu. Rev. Neurosci.*, vol. 47, pp. 277–301, Aug. 2024.

[40] C. Caucheteux and J.-R. King, "Brains and algorithms partially converge in natural language processing," *Commun. Biol.*, vol. 5, p. 134, Feb. 2022.

[41] A. Goldstein, Z. Zada, E. Buchnik, M. Schain, A. Price, B. Aubrey, S. A. Nastase, A. Feder, D. Emanuel, A. Cohen, A. Jansen, H. Gazula, G. Choe, A. Rao, C. Kim, C. Casto, L. Fanda, W. Doyle, D. Friedman, P. Dugan, L. Melloni, R. Reichart, S. Devore, A. Flinker, L. Hasenfratz, O. Levy, A. Hassidim, M. Brenner, Y. Matias, K. A. Norman, O. Devinsky, and U. Hasson, "Shared computational principles for language processing in humans and deep language models," *Nat. Neurosci.*, vol. 25, pp. 369–380, Mar. 2022.

[42] A. Goldstein, E. Ham, S. A. Nastase, Z. Zada, A. Dabush, B. Bobbi Aubrey, M. Schain, H. Gazula, A. Feder, W. Doyle, S. Devore, P. Dugan, D. Friedman, M. Brenner, A. Hassidim, O. Devinsky, A. Flinker, O. Levy, and U. Hasson, "Correspondence between the layered structure of deep language models and temporal structure of natural language processing in the human brain." July 2022.

[43] G. Tuckute, A. Sathe, S. Srikant, M. Taliaferro, M. Wang, M. Schrimpf, K. Kay, and E. Fedorenko, "Driving and suppressing the human language network using large language models," *Nat. Hum. Behav.*, vol. 8, pp. 544–561, Mar. 2024.

[44] F. Pereira, B. Lou, B. Pritchett, S. Ritter, S. J. Gershman, N. Kanwisher, M. Botvinick, and E. Fedorenko, "Toward a universal decoder of linguistic meaning from brain activation," *Nat. Commun.*, vol. 9, Mar. 2018.

[45] C. Kauf, G. Tuckute, R. Levy, J. Andreas, and E. Fedorenko, "Lexical-semantic content, not syntactic structure, is the main contributor to ANN-brain similarity of fMRI responses in the language network," *Neurobiol. Lang. (Camb.)*, vol. 5, pp. 7–42, Apr. 2024.

[46] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 2024. Online manuscript released August 20, 2024.

[47] E. Smith, O. Hsu, R. Qian, S. Roller, Y.-L. Boureau, and J. Weston, "Human evaluation of conversations is an open problem: comparing the sensitivity of various methods for evaluating dialogue agents," in *Proceedings of the 4th Workshop on NLP for Conversational AI*, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2022.

[48] C.-W. Liu, R. Lowe, I. V. Serban, M. Noseworthy, L. Charlin, and J. Pineau, "How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation," 2016.

[49] A. B. Sai, A. K. Mohankumar, and M. M. Khapra, "A survey of evaluation metrics used for NLG systems," 2020.

[50] S. Mehri, J. Choi, L. F. D'Haro, J. Deriu, M. Eskenazi, M. Gasic, K. Georgila, D. Hakkani-Tur, Z. Li, V. Rieser, S. Shaikh, D. Traum, Y.-T. Yeh, Z. Yu,

Y. Zhang, and C. Zhang, "Report from the NSF future directions workshop on automatic evaluation of dialog: Research directions and challenges," 2022.

[51] Y.-T. Yeh, M. Eskenazi, and S. Mehri, "A comprehensive assessment of dialog evaluation metrics," 2021.

[52] J. I. Choi, M. Collins, E. Agichtein, O. Rokhlenko, and S. Malmasi, "Combining multiple metrics for evaluating retrieval-augmented conversations," in *Proceedings of the Third Workshop on Bridging Human–Computer Interaction and Natural Language Processing*, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2024.

[53] D. Khashabi, G. Stanovsky, J. Bragg, N. Lourie, J. Kasai, Y. Choi, N. A. Smith, and D. S. Weld, "GENIE: Toward reproducible and standardized human evaluation for text generation," 2021.

[54] T. Hashimoto, H. Zhang, and P. Liang, "Unifying human and statistical evaluation for natural language generation," in *Proceedings of the 2019 Conference of the North*, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2019.

[55] P. Gupta, S. Mehri, T. Zhao, A. Pavel, M. Eskenazi, and J. Bigham, "Investigating evaluation of open-domain dialogue systems with human generated multiple references," in *Proceedings of the 20th Annual SIGdial Meeting on Discourse and Dialogue*, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2019.

[56] R. Lowe, M. Noseworthy, I. V. Serban, N. Angelard-Gontier, Y. Bengio, and J. Pineau, "Towards an automatic Turing test: Learning to evaluate dialogue responses," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (R. Barzilay and M.-Y.

Kan, eds.), (Vancouver, Canada), pp. 1116–1126, Association for Computational Linguistics, July 2017.

[57] A. B. Sai, M. D. Gupta, M. M. Khapra, and M. Srinivasan, "Re-evaluating ADEM: A deeper look at scoring dialogue responses," *Proc. Conf. AAAI Artif. Intell.*, vol. 33, pp. 6220–6227, July 2019.

[58] C. Callison-Burch, M. Osborne, and P. Koehn, "Re-evaluating the role of Bleu in machine translation research," in *11th Conference of the European Chapter of the Association for Computational Linguistics* (D. McCarthy and S. Wintner, eds.), (Trento, Italy), pp. 249–256, Association for Computational Linguistics, Apr. 2006.

[59] A. Smith, C. Hardmeier, and J. Tiedemann, "Climbing mont BLEU: The strange world of reachable high-BLEU translations," in *Proceedings of the 19th Annual Conference of the European Association for Machine Translation*, pp. 269–281, 2016.

[60] E. Sulem, O. Abend, and A. Rappoport, "BLEU is not suitable for the evaluation of text simplification," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, eds.), (Brussels, Belgium), pp. 738–744, Association for Computational Linguistics, Oct.-Nov. 2018.

[61] J. Deriu, A. Rodrigo, A. Otegi, G. Echegoyen, S. Rosset, E. Agirre, and M. Cieliebak, "Survey on evaluation methods for dialogue systems," *arXiv [cs.CL]*, 2019.

[62] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *J. Acoust. Soc. Am.*, vol. 62, pp. S63–S63, Dec. 1977.

[63] S. F. Chen, D. Beeferman, and R. Rosenfeld, "Evaluation metrics for language models," 1998.

[64] X. Gao, Y. Zhang, M. Galley, C. Brockett, and B. Dolan, "Dialogue response ranking training with large-scale human feedback data," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (B. Webber, T. Cohn, Y. He, and Y. Liu, eds.), (Online), pp. 386–395, Association for Computational Linguistics, Nov. 2020.

[65] A. K. Vijayakumar, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. Crandall, and D. Batra, "Diverse beam search: Decoding diverse solutions from neural sequence models," 2016.

[66] J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan, "A diversity-promoting objective function for neural conversation models," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (K. Knight, A. Nenkova, and O. Rambow, eds.), (San Diego, California), pp. 110–119, Association for Computational Linguistics, June 2016.

[67] Y. Zhang, M. Galley, J. Gao, Z. Gan, X. Li, C. Brockett, and B. Dolan, "Generating informative and diverse conversational responses via adversarial information maximization," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[68] W. Xu, S. Pakhomov, P. Heagerty, E. Horvitz, E. R. Bradley, A. Campbell, J. D. Woolley, A. Cohen, D. Ben-Zeev, and T. Cohen, "Perplexity and proximity: Large language model perplexity complements semantic distance metrics for the detection of incoherent speech." 2025.

[69] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating text generation with BERT," 2019.

[70] G. Datta, N. Joshi, and K. Gupta, "Analysis of automatic evaluation metric on low-resourced language: Bertscore vs bleu score," in *Speech and Computer* (S. R. M. Prasanna, A. Karpov, K. Samudravijaya, and S. S. Agrawal, eds.), (Cham), pp. 155–162, Springer International Publishing, 2022.

[71] S. Xu, W. Xie, L. Zhao, and P. He, "Chain of draft: Thinking faster by writing less," 2025.

[72] C. Beaudoin, É. Leblanc, C. Gagner, and M. H. Beauchamp, "Systematic review and inventory of theory of mind measures for young children," *Front. Psychol.*, vol. 10, p. 2905, 2019.

[73] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, and J. a. Shah, "The llama 3 herd of models,"

[74] R. Stureborg, D. Alikaniotis, and Y. Suhara, "Large language models are inconsistent and biased evaluators," May 2024.

[75] J. E. Eicher and R. F. Irgolič, "Reducing selection bias in large language models," 2024.

[76] M. Sap, H. Rashkin, D. Chen, R. LeBras, and Y. Choi, "SocialIQA: Commonsense reasoning about social interactions," Apr. 2019.

[77] T. Lanham, A. Chen, A. Radhakrishnan, B. Steiner, C. Denison, D. Hernandez, D. Li, E. Durmus, E. Hubinger, J. Kernion, K. Lukošiūtė, K. Nguyen, N. Cheng, N. Joseph, N. Schiefer, O. Rausch, R. Larson, S. McCandlish, S. Kundu, S. Kadavath, S. Yang, T. Henighan, T. Maxwell, T. Telleen-Lawton, T. Hume, Z. Hatfield-Dodds, J. Kaplan, J. Brauner, S. R. Bowman, and E. Perez, "Measuring faithfulness in Chain-of-Thought reasoning," 2023.

[78] M. Turpin, J. Michael, E. Perez, and S. R. Bowman, "Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting," 2023.

[79] S. H. Tanneru, D. Ley, C. Agarwal, and H. Lakkaraju, "On the hardness of faithful Chain-of-Thought reasoning in large language models," 2024.

[80] C. Agarwal, S. H. Tanneru, and H. Lakkaraju, "Faithfulness vs. plausibility: On the (un)reliability of explanations from large language models," 2024.

[81] A. Welivita, Y. Xie, and P. Pu, "A large-scale dataset for empathetic response generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing* (M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, eds.), (Online and Punta Cana, Dominican Republic), pp. 1251–1264, Association for Computational Linguistics, Nov. 2021.

[82] M. J. Traxler and M. A. Gernsbacher, eds., *Handbook of Psycholinguistics*. San Diego, CA: Academic Press, 2 ed., Nov. 2006.

[83] M. K. Tanenhaus, M. J. Spivey-Knowlton, K. M. Eberhard, and J. C. Sedivy, "Integration of visual and linguistic information in spoken language comprehension," *Science*, vol. 268, pp. 1632–1634, June 1995.

[84] J. E. Hanna, M. K. Tanenhaus, and J. C. Trueswell, "The effects of common ground and perspective on domains of referential interpretation," *J. Mem. Lang.*, vol. 49, pp. 43–61, July 2003.

[85] C. G. Chambers, M. K. Tanenhaus, and J. S. Magnuson, "Actions and affordances in syntactic ambiguity resolution," *J. Exp. Psychol. Learn. Mem. Cogn.*, vol. 30, pp. 687–696, May 2004.

[86] S. E. Brennan, "Invited talk: Processes that shape conversation and their implications for computational linguistics," in *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, (Hong Kong), pp. 1–11, Association for Computational Linguistics, Oct. 2000.

[87] G. Aist, J. Allen, E. Campana, L. Galescu, C. A. G. Gallo, S. C. Stoness, M. Swift, and M. Tanenhaus, "Software architectures for incremental understanding of human speech," in *Interspeech 2006*, (ISCA), pp. aper 1869–Wed2FoP.5–0, ISCA, Sept. 2006.

[88] Z. Zada, S. A. Nastase, B. Aubrey, I. Jalon, S. Michelmann, H. Wang, L. Hasenfratz, W. Doyle, D. Friedman, P. Dugan, L. Melloni, S. Devore, A. Flinker, O. Devinsky, A. Goldstein, and U. Hasson, "The "podcast" ECoG dataset for modeling neural activity during natural language comprehension." Feb. 2025.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, p. 2825–2830, Nov. 2011.

[90] T. Dupré la Tour, M. Visconti di Oleggio Castello, and J. L. Gallant, "The voxelwise modeling framework: a tutorial introduction to fitting encoding models to fMRI data." Apr. 2024.

[91] T. Dupré la Tour, M. Eickenberg, A. O. Nunez-Elizalde, and J. L. Gallant, "Feature-space selection with banded ridge regression," *NeuroImage*, vol. 264, p. 119728, 2022.

[92] D. Barrett, *Supernormal Stimuli: How Primal Urges Overran Their Evolutionary Purpose*. New York: W. W. Norton & Company, 2010.

[93] C. Fernyhough and A. M. Borghi, "Inner speech as language process and cognitive tool," *Trends Cogn. Sci.*, vol. 27, pp. 1180–1193, Dec. 2023.